Essays on practical computational methods for operational planning

Essays on practical computational methods for operational planning

Martijn van Brink

This book was typeset by the author using LaTeX and the classisthesis package.

Published by Universitaire Pers Maastricht
ISBN: 978 94 6159 672 7
Printed in The Netherlands by Datawyse Maastricht

ESSAYS ON PRACTICAL COMPUTATIONAL METHODS FOR OPERATIONAL PLANNING

DISSERTATION

to obtain the degree of Doctor at Maastricht University,
on the authority of the Rector Magnificus,
Prof. dr. Rianne. M. Letschert,
in accordance with the decision of the Board of Deans,
to be defended in public
on Friday 10 March 2017, at 12.00 hours

by

Martijn van Brink

Supervisor:

Prof. dr. ir. C.P.M. van Hoesel

Co-Supervisors:

Dr. A. Grigoriev

Dr. T. Vredeveld

Assessment Committee:

Prof. dr. A.J. Vermeulen (chairman)

Dr. A. Berger

Prof. dr. J.A. Gromicho dos Santos (Vrije Universiteit Amsterdam)

Dr. M. Mnich

This research was financially supported by the Graduate School of Business and Economics (GSBE).

To my mother, who has always supported me.

ACKNOWLEDGEMENTS

This thesis represents the results of my research as Ph.D. student. Doing a Ph.D. has been an interesting, instructive and fun experience. It is a period of my life that I will cherish forever. There are a lot of people who have contributed to this remarkable experience, and for that I will be forever grateful. In the following, I would like to thank some of you personally.

First of all, I would like to thank André Berger, Alexander Grigoriev, and Tjark Vredeveld. Without you, I would probably never even have contemplated doing a Ph.D. Even in the final year of my Bachelor studies, I was adamant that I was not going to do a Ph.D. However, this gradually changed during my Master studies. During your challenging and interesting courses we delved deeper into the material, and I soon realized that this sparked my interest in doing research. When eventually the possibility of doing a Ph.D. opened up, I wholeheartedly went for it.

I would also like to thank Alexander and Tjark for being my supervisors and Stan van Hoesel for being my promoter. You had the belief in me that I could successfully complete a Ph.D. and offered me a position as Ph.D. student. You also realized that I had a strong interest in the practical side of research, i. e., in implementing and evaluating the developed methods on realistic instances. From day one the goal of my research was to consider problems from both a theoretical and practical perspective, and this goal did not change during my research. You gave me a lot of freedom in selecting the problems to work on, a freedom that I gladly used (some might even say abused). Although this may have resulted in more time being needed to complete this thesis, it has also greatly benefited my personal development.

I thank my co-authors Gergely (Greg) Csapó and Ruben van der Zwaan. Greg, without you I would have never known about the Subcontractor Scheduling problem. Even more important (at least to me), I would not have known what

my third problem to research would be. What seemed like a simple problem resulted in many interesting discussion sessions. Ruben, when we started on the Container Premarshalling problem, it was not even clear if our intended approach could actually work, let alone give good results. Whenever we got stuck, you could always quickly pinpoint which step of our approach was causing the issue and formulate an alternative solution. This resulted in our paper being accepted for ESA 2014 (and a nice trip to Wrocław, Poland for me).

I would also like to thank the members of my assessment committee, Dries Vermeulen, André Berger, Joaquim Gromicho dos Santos, and Matthias Mnich for your time and thorough reading of my thesis.

I thank the secretaries of the Quantitative Economics department, Karin van der Boorn, Haydeé Hallmanns, and Yolanda Paulissen. Not only did you take care of all administrative stuff, you never complained when we invaded the secretary office for one of our (many) tea breaks.

I also thank all my fellow Ph.D. students of the fourth floor. Not only did you help in creating a great working environment, you also provided pleasant distractions during tea and lunch breaks. Special thanks go to Greg, Thomas Götz, Andrey Kateshov, Vincent Kreuzen, Abhinaba Lahiri, Jan Lohmeyer, Tim Oosterwijk, Marc Schröder, Veerle Timmermans, and Anna Zseleva for the many activities outside of work: having dinner together, going to pub quizzes, game nights, movie nights, going to the Efteling, and much much more.

I would also like to thank my office mates Greg, Jan, Hans Ensinck, Cyriel Rutten, and Birol Yüceoğlu. You were all great colleagues to both work with and to share an office with. When needed, you were there for both scientific and non-scientific discussion. You gave a heart and soul to the office, creating both a fun and productive environment.

A task I enjoyed more than expected was being a tutor. For a large part the credits have to go to Alexander, who was the course coordinator for the majority of the courses that I tutored. By providing an extensive tutor manual and organizing regular tutor meetings, you made it easy to prepare for the tutorials. Also, if we had any questions you were more than willing to help us. You also ensured that correcting the exams was as much fun as possible, by providing snacks and beer

(to reassure the reader: the beer was only provided after we finished correcting the exams).

Finally, I would like to thank my family, and especially my mother. If it was not for you, I would not have switched to studying Econometrics and Operations Research at Maastricht University. I know that I can always count on your support, no matter which decisions I make. For that I will be forever grateful.

CONTENTS

1

INTRODUCTION

This thesis discusses three problems from the field of operations research, all based on real-life problems. These problems are considered from both a theoretical and practical perspective. With respect to the theoretical perspective one can think of the difficulty (complexity) of the problem, or the worst case performance of algorithms (performance guarantee). From the practical perspective we consider the actual performance of algorithms on realistic instances (both in size and construction), inspired by real-life instances.

The main focus of this thesis is on the viewpoint of the practitioners, for whom the practical results are more interesting. It is very hard to sell an algorithm with performance guarantee 2 to a practitioner, i. e., an algorithm that for each possible instance "only" guarantees a solution that is at most twice as expensive as the optimal value. However, an algorithm that improves the best known solution by "only" a (few) tenth(s) of a percent can already save millions, making it potentially very interesting for practitioners.

By definition it is impossible to capture all details and complexities of a problem, implying that always an abstraction of the actual problem is considered. As we focus on the practical side, we try to keep the studied problems as close to the original problem as possible. Observe that this might result in an abstracted problem which is more difficult to study, or classify in terms of difficulty. It might also be harder to construct elegant and efficient algorithms that can be easily analyzed. However, problems are sometimes oversimplified, making the obtained results less useful for practitioners. By considering as many details as possible we hope to construct algorithms that perform better on instances of the actual problem.

Observe that we are not saying that our viewpoint is the best one, or even the only proper one; it is just the viewpoint we decided to focus on. Looking at different levels of abstractions can also have it advantages. It might be possible to identify a certain sub-problem or combination of sub-problems that greatly affect the difficulty of the problem. Also, efficient algorithms for sub-problems can possibly be incorporated into solutions for the actual problem.

With respect to solutions currently used in practice, we believe that improvements can be obtained in two steps. First, by applying the currently known machinery of solution approaches on less abstracted problems, "easy" gains can be obtained. In the second step, specific research on a detailed (sub-) problem can further increase the gains. Hence, we believe improvements can be obtained by improving the current (general) machinery of solution approaches, and by tailoring methods to specific problems.

As already stated, we focus on the practical perspective, keeping as many details as possible. Therefore, the methods presented in this thesis pertain to the second step mentioned in the previous paragraph. We thus do not consider three problems with overlapping characteristics, nor do we present a single method or approach that can be easily applied to all three (or even more) problems. What we will do is consider three operations research problems which we think are interesting and deserving of attention, and develop algorithms that are tailored towards the specifics of each problem.

In the remainder of this section we give a short overview of several methods that are employed in this thesis. Thereafter we go into more detail about the three problems that are discussed.

1.1 COMBINATORIAL OPTIMIZATION

All three problems discussed in this thesis can be classified as a combinatorial optimization problem. In combinatorial optimization one is asked to find the best possible solution among a number of alternatives. The quality of a solution is measured by a cost function that assigns a value to each solution. More formally, a combinatorial optimization problem is specified by a set of problem instances, and each problem instance only allows finitely many, or countably infinitely many, solutions. A combinatorial optimization problem is either a min-

imization or maximization problem, depending on the objective of the problem. The set S of solutions and cost function f are usually not given explicitly, but by an implicit description.

1.2 COMPUTATIONAL COMPLEXITY THEORY

Some problems are easy to solve, while other problems turn out to be difficult. Thereto, we are interested in determining the running time needed to obtain an optimal solution. More precisely, given an instance $\mathcal{I}$ for problem Π, with encoding length $|\mathcal{I}|$, we are interested in the running time with respect to $|\mathcal{I}|$. If for an algorithm $\mathcal{A}$ there exists a polynomial $p(.)$ such that for all instances $\mathcal{I}$ the running time is bounded by $p(|\mathcal{I}|)$, we say that $\mathcal{A}$ is a polynomial time algorithm.

To formalize this notion, we introduce *decision problems*. A decision problem is a problem where given an instance $\mathcal{I}$, the output is either 'yes' or 'no'. Decision problems for which a polynomial time algorithm exists that always gives the correct answer, belong to problem class P. Another problem class is NP. A decision problem Π belongs to the class NP if for all instances $\mathcal{I} \in \Pi$ with solution 'yes', there exists a *certificate* x of size polynomial in $|\mathcal{I}|$, for which one can verify in polynomial time that x is indeed a certificate for $\mathcal{I}$. Clearly $P \subseteq NP$, and one of the biggest open questions in the field of computer science is whether $P = NP$. The consensus though seems that this does not hold. Hence, for all problems in NP that have not been proven to be also in P, currently no polynomial time algorithm exists that always obtains the correct answer.

For a more thorough treatment of computational complexity theory, the reader is referred to for instance [36] and [60].

1.3 LINEAR PROGRAMMING

Linear programming is a widely used field of optimization. A minimization problem Π (the results are symmetric for maximization problems) can often be formulated as follows

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \geqslant b \\ & x \geqslant 0, \end{aligned} \tag{LP}$$

where x is a vector of variables, $c^T x$ the (linear) objective function, and the rows of $Ax \geqslant b$ the (linear) constraints. The constraints define the *feasible region*, i. e., the set of points that satisfy all constraints. Note that an optimal solution need not exist. If two constraints are inconsistent (for instance, $x_1 \geqslant 3$ and $x_1 \leqslant 2$) then the feasible region is empty. In this case, we say that LP is *infeasible*. If the feasible region is unbounded in the direction of the objective function, no optimal solution can be obtained. In this case, we say that LP is *unbounded*.

A very popular method for solving linear program formulations is the *simplex method*, developed by George Dantzig. Other methods are for instance the ellipsoid method and interior point methods. Where these last two methods have a polynomial worst case running time, the simplex method has an exponential worst case running time. However, this is caused by some artificial instances; in practice, the simplex method is efficient, and performs among the best.

For a more thorough treatment of linear programming, the reader is referred to for instance [18] and [25].

1.4 BRANCH-AND-BOUND

Branch-and-Bound is a commonly used solution technique to solve combinatorial optimization problems. We only consider minimization problems, the algorithm is symmetric for maximization problems. A branch-and-bound algorithm iteratively partitions the solution space into smaller subsets. For each subset a lower bound is determined for the optimal solution value in that subset. If for a subset the lower bound is at least the value of the best feasible solution found so far, then this subset is discarded, as it will not contain any solution strictly better than the best one found so far. The remaining subsets are partitioned until all subsets are discarded. After al subsets are discarded, the best found feasible solution is by definition the optimal solution. There are three main ingredients for any branch-and-bound algorithm: (1) a branching rule, that partitions each subset into smaller subsets; (2) a subset selection rule, that determines which subset is considered next; and (3) a means to determine a lower bound for a specific subset.

For a more thorough treatment of branch-and-bound and related methods, the reader is referred to for instance [59] and [61].

1.5 APPROXIMATION ALGORITHMS

If a problem is NP-hard, it is unlikely that a polynomial time algorithm that finds an optimal solution exists. One way to resolve this problem is to look for approximation algorithms that find good, but not necessarily optimal, solutions in polynomial time. A special type of approximation algorithms is the $\alpha-$approximation algorithm, that gives a guarantee on the quality of the obtained solution. An algorithm $\mathcal{A}$ is an $\alpha-$approximation algorithm for minimization problem Π (the definition is similar for maximization problems) if for all instances $\mathcal{I}$ of problem Π we have that $\mathrm{OPT}(\mathcal{I}) \leqslant \mathcal{A}(\mathcal{I}) \leqslant \alpha \cdot \mathrm{OPT}(\mathcal{I})$, where $\mathcal{A}(\mathcal{I})$ and $\mathrm{OPT}(\mathcal{I})$ denote the value obtained by algorithm $\mathcal{A}$ and the optimal value for instance $\mathcal{I}$, respectively. The value α is called the performance guarantee, or worst-case ratio, of algorithm $\mathcal{A}$. Note that the performance guarantee is a worst-case bound that has to hold for all instances. For a large group of instances the algorithm might perform much better than the performance guarantee indicates, maybe even providing the optimal solution.

For a more thorough treatment of approximation algorithms, the reader is referred to for instance [80].

1.6 LOCAL SEARCH

The approximation algorithms described in Section 1.5 only determine a single solution. A next step is to consider other solutions that improve upon this initial solution. One way to achieve this is by a method called local search. Starting from the initial solution, the local search algorithm keeps moving to different feasible solutions until some stopping criterion is met. When determining to which solution to move, not all possible solutions are considered, but only solutions that are in some sense close to the current solution. These solutions constitute the *neighborhood* of the current solution. More formally, for an instance $\mathcal{I}$ of combinatorial optimization problem Π with feasible set of solutions S, let $\mathcal{N} : S \to 2^S$ define for each $s \in S$ a set $\mathcal{N}(s) \subseteq S$ of solutions that are in some sense close to s. Each $s' \in \mathcal{N}(s)$ is called a neighbor of s.

The simplest form of local search is called *iterative improvement*. Starting from the initial solution, a neighboring solution is selected iteratively, but only if the neighboring solution is a strict improvement on the current solution. The local

search procedure stops if for the current solution no neighboring solution is strictly better. Such a solution is called a local optimum with respect to $\mathcal{N}$.

For a more thorough treatment of local search methods, the reader is referred to for instance [1].

1.7 ABSTRACTS

In this section we give an outline for the remainder of this thesis, and we go into more detail about the three problems that are discussed.

1.7.1 Express Delivery

In Chapter 2, which is based on Van Brink, Grigoriev, and Vredeveld [77], the Express Delivery problem is discussed. For this problem we are given a set of commodities and a set of locations. Each commodity is specified by a start location, end location, volume, delivery time, and a set of allowed routes. For each pair of locations we are given the distance and travel time between them. The commodities are transported with trucks, which all have the same capacity. We assume that a truck is only used on a single connection between two locations. The goal is to find a route for each commodity that allows timely delivery. For each connection in the selected route, the time at which it is traversed also needs to be specified. These times need to be consistent with the order of the connections in the route. After all routes are selected, it is straightforward to determine how many trucks are needed at each time point for each connection. The objective is to minimize the transportation cost. These cost only stem from operating the trucks.

The express delivery problem is faced by many shipping companies. The three largest shipping companies in the world are UPS, DHL, and FedEx [70]; UPS and FedEx together already deliver over 28 million packages and documents per day [29, 72].

The express delivery problem contains elements from several well known problems, for instance the capacitated multicommodity flow problem, the multicommodity flow over time problem, and the time-space fixed-charge network flow problem. The main difference with the capacitated multicommodity flow prob-

lem is that the express delivery problem additionally contains transit times on the arcs. Compared to the multicommodity flow over time problem, there is an additional need to "buy" capacity. If we compare the time-space fixed-charge network flow problem and the express delivery problem, we primarily observe a difference in the encoding of the arcs. For a standard space-time expansion each arc is identified by four parameters: the start node, end node, start time, and end time. Hence, it is possible to "decide" on the transit time needed for an arc. For the express delivery problem the transit time for an arc is fixed, and only depends on the start and end node. This means that each arc can already be specified by three parameters: the start node, end node, and start time. This allows for a more compact encoding, and thus different solution approaches.

We first consider the complexity of the express delivery problem. We show that the problem is not only NP-hard, but also hard to approximate within a factor that is logarithmic in the number of commodities. With respect to the allowed routes we consider three settings. In the first setting all possible routes are allowed, while in the other two settings, only routes that have certain intermediate locations, called hubs, are allowed. For each instance three hubs are determined, greatly decreasing the number of allowed routes. For the solution methods we consider an ILP formulation, two LP-based rounding algorithms, two constructive heuristics, and three local search neighborhoods. We evaluate these methods on 240 instances that are based on problems from TSPLIB [64]. TSPLIB is the most used library of instances for the Traveling Salesman Problem and related problems. Besides the TSPLIB problem on which it is based, the instances also vary on the number of commodities. With respect to the results, one would expect that the best solutions are obtained in the setting where all possible routes are allowed. Especially for the ILP formulations this is not always true. For the bigger instances it is often not even possible to obtain a feasible solution, while for the smaller instances using a restricted set of allowed routes sometimes gives better results when only a limited amount of time is available. Generally, we have that for the smaller instances the ILP formulation slightly outperforms one of the local search neighborhoods, but for the larger instances the local search neighborhood clearly outperforms the ILP formulation.

1.7.2 Container Premarshalling

In Chapter 3, which is based on Van Brink and Van der Zwaan [78], the Container Premarshalling problem is discussed. For this problem we are given a container yard filled with containers. The container yard consists of several stacks, each with the same height. For each container we are given a priority level and a position. The position is determined by the stack in which the container is placed and the height at which it is positioned. The containers are moved with the help of a single crane, that can move one container at a time. The goal is to find a feasible sequence of moves that transforms the lay-out of the container yard into a desired lay-out. A move is specified by the priority level of the container that is moved, as well as its origin and destination stack. A move is feasible if (1) the priority level of the top container of the origin stack matches the specified priority level, and (2) the destination stack is not full. For the desired lay-out there are two options, called Priority Stacking and Configuration Stacking. For Priority Stacking all lay-outs such that no container with a lower priority is placed on top of a container with a higher priority are accepted, while for Configuration Stacking there is a single pre-specified desired lay-out. The objective is to obtain a desired lay-out in the shortest time possible. As the time that is needed to move a container is dominated by the timed needed to pick up or drop off a container, this is equivalent to minimizing the number of moves.

The container premarshalling problem deals with the transshipment of containers. The three largest transshipment hubs in the world are the ports of Singapore, Shanghai, and Shenzhen. Together, these ports transshipped roughly 54.5 million TEU (Twenty Feet Equivalent Unit) in 2013 [62]. This corresponds to a volume of roughly 2 billion cubic meter.

Although operations at container terminals have been extensively studied, not much research has been directed towards the container premarshalling problem. With respect to the complexity of the problem, it was only known that Priority Stacking is NP-hard for arbitrary stack height (more precisely, a stack height of at least the total number of containers). The focus of most papers on the container premarshalling problem is on developing fast heuristics. We are aware of only a single paper that focuses on an exact algorithm, which is evaluated on only two instances.

We first consider the complexity of the premarshalling problem, and show that both PRIORITY STACKING and CONFIGURATION STACKING are NP-hard for a fixed stack height. Next, we present an exact algorithm based on branch-and-cut. We introduce the (primal and dual) LP formulations and the separation oracle, and explain how they are incorporated into our branch-and-price framework. Finally, we evaluate our method on 960 randomly generated instances. These instances vary on the number of priority levels, the number of stacks, the height of the stacks, and the fill grade of the container yard. Hence, to the best of our knowledge, we are the first to extensively evaluate an exact algorithm for the container premarshalling problem. Out of all the instances, 98.4% are solved to optimality within one hour, 93.2% within one minute, and 70.8% within one second. One drawback of our method is that it either gives the optimal solution, or no solution at all. By slightly adjusting our approach it should be possible to also find near-optimal solutions.

1.7.3 Subcontractor Scheduling

In Chapter 4, which is based on Van Brink, Csapó, and Van der Zwaan [76], the Subcontractor Scheduling problem is discussed. For this problem we are given a set of jobs and an additional resource, called the subcontractor. For each job, which has at its disposal a private machine, we are given a release date, processing time, and weight. If the subcontractor is assigned to assist a certain job, then for the duration of the assignment the processing speed of that job is doubled. The goal is to find an assignment for the subcontractor that specifies for each time point to which job it is assigned. This assignment has to be feasible, i. e., the subcontractor can only be assigned to one job at a time, cannot start assisting a job before its release date, and can no longer assist a job after its completion time. Observe that for this setting the completion time of a job is decreased by the same amount as the time that the subcontractor is assigned to it. If we define this decrease of the completion time as the savings of a job, then the objective is to maximize the weighted sum of savings. Note that in this case, this corresponds to the more widely used measure of minimizing the weighted sum of completion times.

Subcontracting is used in many industries. Traditionally, it has been primarily used in construction. Nowadays, subcontracting is also extensively applied at in-

formation technology companies, potentially even surpassing the usage in construction. Observe that using subcontractors is not without risk, as the case of Boeing's Dreamliner demonstrates [31].

As only the schedule of the subcontractor is of interest, this problem can be viewed as a single machine scheduling problem. More precisely, the subcontractor scheduling problem resembles the problem of minimizing the (weighted) sum of completion times. For the problem without release dates it is a known result that scheduling the jobs in non-decreasing order of their processing time (if jobs have no weight) or according to Smith's ratio (if jobs have weights) gives the optimal solution. For the situation with release dates the problem is already NP-hard, even if preemption is allowed. The main difference with the subcontractor scheduling problem lies in the fact that for the subcontractor scheduling problem, the remaining processing time depends on the time at which the subcontractor is assigned. Hence, it is not even guaranteed that the subcontractor is assigned to all jobs. The subcontractor scheduling problem furthermore resembles the problem of scheduling jobs with decay rates. If the decay rates can differ per job, it is known that the problem is NP-hard. Note that for the subcontractor scheduling problem all jobs have the same decay rate.

Initially we consider four heuristics, called Proctime, Weight, Smith, and Profit. To determine the next job that the subcontractor will assist, we respectively take the job with the shortest remaining processing time, the highest weight, the highest Smith ratio, or the highest value for weight times remaining processing time. Note that Proctime an Smith resemble the rules that give the optimal solution for respectively minimizing the sum and weighted sum of completion times on a single machine. For the case with equal weights and no release dates it is a known result that Proctime gives the optimal solution. However, in our setting with arbitrary weights it clearly has the worst performance guarantee of the four heuristics. The heuristic with the best guarantee is Weight. This is somewhat surprising, as it only takes one dimension of the input (weight) into account, while Smith and Profit take two dimensions (processing time and weight) into account. We evaluate the heuristics on 9000 randomly generated instances. The instances vary on the number of jobs and the distributions for both the processing time and the weight. As only considering the weight gives the best guarantee, we add additional heuristics, mostly variants of Smith and Profit, that give a higher

importance to the weight. With respect to release dates, preemption, and the ordering of the jobs we also consider several settings. The results of this study confirm the theoretical results: for almost all instances and settings Weight performs best, while Proctime performs badly. Compared to scheduling the jobs in order of their index (which is identical to scheduling the jobs in a random order), Proctime only scores slightly better. With respect to allowing preemption, we observe better solutions, but at the expense of an increased running time.

2

EXPRESS DELIVERY

2.1 INTRODUCTION

Freight transportation can account for ten to fifteen percent of the Gross National Product (GDP) of a country, and it is most likely that these figures will increase in the future [20, 24]. Freight transportation is usually subdivided into two main parts, namely (full) truckload (FTL) shipping and less than truckload (LTL) shipping. FTL shipping is used for large freight volumes, while LTL shipping is used for small volumes. In case of LTL shipping, freight from several sources is consolidated to achieve a large reduction in cost. A special case of LTL shipping is express delivery, where there is a hard and tight deadline to deliver the packages. Normally, packages need to be delivered within 24 or 48 hours.

The delivery process consists of three phases. In the first phase, the packages are picked up from their origin, and delivered at a transshipment point called a *terminal*. In the second phase, the packages are transported between terminals, and in the third and final phase, the packages are delivered from the terminal to their destination. In this chapter we consider the second phase, the transportation of packages between the terminals. In this phase, the packages are not considered individually, but all packages that have the same start and end terminal are grouped together in a single commodity. Express delivery companies generally require that all packages in a commodity are sent via the same route. In the literature this is called *non-bifurcated*.

Within a country the packages are transported by trucks. We assume that between every pair of terminals there are enough trucks available to transport the requested amount. For each truck that is used a certain cost is incurred. This cost is due to the operating of the truck between two terminals.

2.1.1 Problem Description

The problem of routing the packages between terminals, further referred to as DELIVERY, is defined as follows. The road network over which the packages are transported, is given by a directed graph $D = (N, A)$, where the set of nodes N are the terminals and the set of arcs A the direct connections between the terminals. The packages are not considered individually, but all packages with the same start and end location are combined into a single *commodity*. The volume of the commodity is equal to the sum of volumes of the packages. Let K denote the set of commodities that need to be routed through the network, and let s_k, t_k, and v_k denote the start terminal, end terminal, and volume of commodity $k \in K$. We assume that all packages, and thus also all commodities, are available for transportation at time 0. Moreover, for each commodity $k \in K$ there is a set R_k of prespecified (s_k, t_k)-routes. The deadline for delivery is denoted by T_k. Without loss of generality, we assume that the set R_k contains only routes that allow the timely delivery of the commodity. If for all k, R_k consists of all possible (s_k, t_k)-routes in D, then we say that the routes are *unrestricted*.

One or more trucks with capacity c transport the commodities over a given connection (u, v). We assume that an infinite number of trucks is available at each terminal. The cost of transporting along connection (u, v) is d_{uv} per truck. The *transit time* needed to traverse connection (u, v) is denoted by $\tau_{uv} \geqslant 0$. This means that if a truck leaves terminal u at time t, it arrives at terminal v at time $t + \tau_{uv}$. We say that connection (u, v) is used at time t when the truck leaves terminal u at time t. Without loss of generality we assume that the transit times are integral. Two or more commodities that are transported over the same connection may share one or more trucks to save costs. This is only possible when the commodities traverse the connection at the same time. Therefore, in a feasible solution we not only have to specify the selected route for each commodity, but also at which time each connection of the selected route is used. Of course, the times at which the connections in a route are traversed need to be consistent with the order in which they appear in the route and the transit times. Since all commodities need to be delivered by the maximum deadline, at most $T_{max} = \max_{k \in K}\{T_k\}$ time units need to be considered. As the following example shows, the timing constraints impose a restriction on how commodities can be combined.

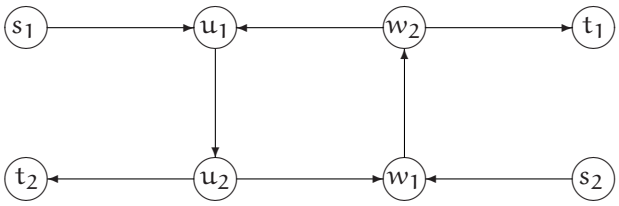

Figure 1: Instance with two commodities that cannot be combined on all mutual connections

Example 2.1. Consider the network in Figure 1. Two commodities need to be routed: commodity 1 from s_1 to t_1 and commodity 2 from s_2 to t_2. Assume that both commodities fit together in one truck and that the transit time for all connections is equal to one. Both commodities need to be transported over connections (u_1, u_2) and (w_1, w_2). However, for commodity 1 connection (u_1, u_2) appears before (w_1, w_2) in the route, whereas for commodity 2 connection (u_1, u_2) appears after (w_1, w_2). Hence, the two commodities can be combined into one truck on at most one of these two connections. This is in contrast with the multicommodity flow setting, where it would be possible to combine the two commodities on both connections.

The flow problem that we consider is non-bifurcated. That means that all packages corresponding to one commodity need to be routed over the same route at the same time. As it might well be the case that the total volume of these packages is more than the capacity of a truck, we do allow for one commodity to be split over several trucks as long as the trucks drive along the same connections and at the same times. Therefore, when subset of commodities K' is routed along connection (u, v) at the same time, we only need $\lceil \sum_{k \in K'} v_k / c \rceil$ trucks to transport these commodities.

The goal of the problem is to deliver all commodities before their deadline, at the minimum cost possible. We only consider the cost incurred by the trucks traversing the connections. The cost of for instance transferring a commodity

from one truck to another, or temporarily storing a commodity at a terminal, are disregarded. If we let y^t_{uv} denote the number of trucks that use connection (u, v) at time t, the total cost is equal to $\sum_{(u,v)\in A} \sum_{t\in T} d_{uv} y^t_{uv}$.

2.1.2 Related Literature

The problem at hand combines elements from several well-studied problems. It resembles the time-space fixed-charge network flow problem, described in for instance [50]. There is however a major difference with respect to the encoding of arcs. For a standard space-time expansion each arc is specified by a quadruple (i, j, t, s). This indicates leaving node i at time t, and arriving at node j at time s. This means that it is possible to decide on the transit time for flows. This is in contrast with the DELIVERY problem, where for a given pair of nodes (i, j) the transit time is fixed, namely τ_{ij}. This also allows for a more compact encoding of the arcs. Instead of a quadruple, only a triple (i, j, t) is needed. This triple specifies that node i is left at time t, and that node j is reached at time $t + \tau_{ij}$. The DELIVERY problem furthermore resembles a capacitated multicommodity flow problem, but with additionally transit times on the arcs, or a multicommodity flow over time problem [66], but with additionally the need to buy capacity.

The DELIVERY problem can be seen as a variant of the minimum cost multicommodity flow problem (MCF), which is extensively studied. For the MCF we are given a graph $G = (V, E)$, capacity $c(u, v)$ and unit flow cost $a(u, v)$ for each $(u, v) \in E$, and k commodities (s_i, t_i, d_i), where s_i is the source node, t_i the sink node, and d_i the demand of commodity i. The goal is to find valid flow values $f_i(u, v)$ for each commodity i and $(u, v) \in E$ that minimizes the cost. The flow values are valid if demand d_i is sent from s_i to t_i for all commodities, while respecting the capacities, i. e., the sum of all $f_i(u, v)$ is at most $c(u, v)$ for all $(u, v) \in E$. The cost incurred by edge (u, v) is the total flow over that edge multiplied by the unit flow cost $a(u, v)$, and the total cost is the sum of all edge cost. For the integral MCF the amount of flow for a commodity over an edge is restricted to be integral. Even, Itai, and Shamir [27] showed that deciding if an integral solution exists is NP-complete even for two commodities. If for each commodity exactly one route needs to be selected, the problem is called non-bifurcated or unsplittable. This setting is considered in, e. g., [6].

One difference between MCF and DELIVERY is the cost structure. For MCF the cost obtained by an edge is linear in the amount of flow over that edge, i. e., one unit of additional flow on edge e leads to a cost increase equal to the unit flow cost $a(e)$. For DELIVERY the cost is determined by the number of trucks used on a connection, which is not linear in the total volume on that connection. Two problems that also have this characteristic are the fixed-charge MCF and the capacitated MCF, also known as the network design problem (NDP).

For the fixed-charge MCF a fixed cost has to be paid when an arc is used, next to the flow cost. Although this problem is well studied, exact methods can only solve small instances, whereas real-life instances remain intractable. Furthermore, the two natural integer linear program formulations, the arc-based and the path-based formulation, suffer from a high integrality gap. Hence, not only can it be difficult to find a feasible solution (under capacity restrictions), even determining its quality can be challenging [44]. Crainic, Frangioni, and Gendron [21] apply Lagrangian relaxation to obtain better bounds, while Frangioni and Gorgone [35] apply bundle methods, exploiting the structure of the cost function, to obtain tight lower bounds. Crainic, Gendreau, and Farvolden [23] apply a tabu search framework to obtain upper bounds. Crainic and Gendreau [22] extend this approach by adding parallelization. Ghamlouche, Crainic, and Gendreau [42] consider cycle based neighborhood structures. Rodríguez-Martín and Salazar-González [65] consider a heuristic approach using local branching. Neighborhoods are explored using a MIP solver, where the neighborhood consists of all solutions satisfying so-called local branching constraints. These constraints indicate how many binary variables can switch value. Gendron and Larose [40] describe a branch-and-price-and-cut algorithm which can also solve instances with a large number of commodities. Katayama, Chen, and Kubo [49] apply an approximate iterative method to solve the path-based formulation. At each iteration the arc capacities are adjusted, and additional constraints and variables are added through row and column generation. Hewitt, Nemhauser, and Savelsbergh [44] consider both the arc-based and path-based formulation. The arc-based formulation is used to obtain primal solutions, by exploring very large neighborhoods via integer linear programs. The path-based formulation, amended with cuts discovered during the solving of the integer linear programs, is used to obtain tight dual bounds.

While for the fixed-charge MCF only a single amount of capacity can be used for each arc, for the capacitated MCF multiple amounts of capacity, generally called facilities, can be purchased. Generally several types of facilities are available, with different capacity levels and purchasing prices. Note that a feasible solution always exists, since an unlimited amount of capacity can be installed on each arc. Magnanti, Mirchandani, and Vachani [58] consider the capacitated MCF without flow cost, i. e., the only cost stem from installing facilities on the arcs. They consider networks with either two or three nodes, and in both cases they characterize the convex hull by giving facet defining inequalities. Bienstock, Chopra, Günlük, and Tsai [9] also consider the capacitated MCF without flow cost. They consider the standard integer linear program formulation and project out the continuous flow variables, after which metric inequalities are added. Frangioni and Gendron [33] consider a reformulation of the standard integer linear program to the capacitated MCF, for which extended linking inequalities are derived. This approach leads to a pseudo-polynomial number of variables and constraints. To this formulation a row and column generation approach is applied. Berger, Gendron, Potvin, Raghavan, and Soriano [7] consider the capacitated MCF with a single source node, and provide a tabu search framework. Gendron, Potvin, and Soriano [41] extend this approach by providing an additional heuristic to obtain a start solution and several diversification strategies. Frangioni and Gendron [34] describe a structured Dantzig-Wolfe decomposition. This decomposition, which is an extension of the standard Dantzig-Wolfe decomposition, exploits the structure of the pricing problem. In [4, 12, 63, 79] the polyhedral structure of several variants for the NDP are considered and several classes of valid inequalities are studied.

For a more detailed overview of models and algorithms for the fixed-charge and capacitated MCF, the reader is referred to [19, 20, 39].

Although the capacitated MCF closely resembles DELIVERY, it does not exhibit the transit times. A problem that does contain the time component is the (discrete) maximal dynamic flow problem, as introduced by Ford and Fulkerson [32]. Consider a network with T time periods and a capacity and traversal time for each arc, the goal is to maximize the amount of flow which can be sent from source node s to sink node t in the T time periods. The authors also provide an efficient algorithm that determines the maximum amount of flow. Since Ford

and Fulkerson introduced this problem, a vast amount of research has been directed towards *dynamic flows*, as the problem is called in the literature. See for instance [51, 53, 66] and references therein for an overview of developments on dynamic flows. An extension of the maximal dynamic flow problem is the quickest transshipment problem. Given several sources and several sinks, the goal is to sent an exact amount from the sources to the sinks in minimum time. Hoppe and Tardos [45] were the first to give a polynomial time algorithm for the quickest transshipment problem with multiple sources and multiple sinks. Köhler and Skutella [52] consider the quickest transshipment problem with a single source and sink, but with transit times that depend on the amount of flow on that arc. The authors provide a 2-approximation algorithm, and show that the quickest transshipment problem with load dependent transit times is strongly NP-hard and also APX-hard. Fleischer and Tardos [30] consider several variants of dynamic flow problems, and extend polynomial time algorithms for discrete-time models to also work on continuous-time models.

2.1.3 Organization

In Section 2.2 we review the complexity results for DELIVERY. In Section 2.3 we discuss two mixed integer program formulations for DELIVERY and two LP-based rounding algorithms. In Section 2.4 we present several constructive heuristics, that obtain solutions from scratch, and improvement schemes, that try to improve feasible solutions. In Section 2.5 we consider the experimental results obtained by applying the described methods on randomly generated instances. Finally, conclusions are drawn in Section 2.6.

2.2 COMPLEXITY

In this section we investigate the complexity of the DELIVERY problem. We first establish NP-hardness, and in doing so we immediately obtain hardness of approximation. Thereafter, we consider two special cases of the DELIVERY problem. For the first case we restrict the underlying graph, while for the second case the number of commodities is restricted

Theorem 2.2. *The* DELIVERY *problem is NP-hard.*

Proof. We make a reduction from SET COVER, which is well known to be NP-complete [48]. The problem SET COVER is defined as follows. Given an universe $\mathbb{U} = \{u_1, \ldots, u_n\}$ and a family of its subsets, $\mathbb{S} = \{S_1, \ldots, S_m\} \subseteq 2^{\mathbb{U}}$ such that $\bigcup_{S_j \in \mathbb{S}} S_j = \mathbb{U}$, the goal is to find a sub-family $\mathbb{C} \subseteq \mathbb{S}$ of minimum cardinality that covers the whole universe, i. e., $\bigcup_{S_j \in \mathbb{C}} S_j = \mathbb{U}$.

Given an instance of SET COVER, we construct the following instance for the unrestricted DELIVERY problem. There are three types of terminals: $|U|$ element terminals, $|S|$ set terminals, and one goal terminal, denoted by t. Each element terminal corresponds uniquely to an element $u_i \in U$ and is therefore denoted by u_i. For each subset $S_j \in S$, there is one set terminal, denoted by S_j. There is a direct connection from every set terminal S_j to goal terminal t. Traversing these connections costs one unit. Furthermore, there is a direct connection from element terminal u_i to set terminal S_j if and only if $u_i \in S_j$. These connections have zero cost. The transit time of all connections is one time unit. For each element u_i, there is a commodity with start terminal u_i and end terminal t. The volume of each of these commodities is one unit. The capacity of the trucks is large enough so that all commodities can be combined in a single truck, e. g., the capacity is at least $|U|$. See Figure 2 for an example.

The deadline for each commodity is 2. As all connections have a transit time of one time unit, this implies that each commodity needs to be transported over at most two connections. As all start (element) terminals only have a direct connection towards set terminals, these connections need to be traversed in the first time slot. During the second time slot the commodities are transported from the set terminals to the goal terminal. As the truck capacity is large enough, all commodities that have been sent to the same set terminal can all be combined in one truck towards the goal terminal. As an element terminal u_i is only connected to set terminal S_j if the corresponding element u_i is contained in the corresponding subset S_j, the sets corresponding to the set terminals from which a truck is sent to the goal terminal form a set cover. Moreover, as costs only stem from sending a truck from a set terminal to the goal terminal, the size of the set cover is equal to the cost of the corresponding DELIVERY solution, and vice versa. □

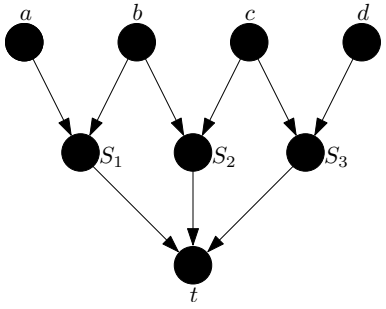

Figure 2: DELIVERY instance based on SET COVER problem $S_1 = \{a, b\}$, $S_2 = \{b, c\}$, $S_3 = \{c, d\}$

Because of the one-to-one correspondence between the solutions for SET COVER and DELIVERY, and the inapproximability result of Alon, Moshkovitz, and Safra [2], we arrive at the following corollary.

Corollary 2.3. *There is a constant $C > 0$ such that it is NP-hard to approximate* DELIVERY *within a factor $C \ln |K|$.*

As the general problem is already NP-hard, we consider the case where the underlying network is restricted to a path.

Theorem 2.4. *The* DELIVERY *problem, where the underlying network is restricted to a path, is strongly NP-hard.*

Proof. We reduce from 3-PARTITION, which is strongly NP-complete [37]. In 3-PARTITION we are given weights $w_1, \ldots, w_{3n}$ such that $\sum_i w_i = nB$. Without loss of generality, we may assume that $\frac{B}{4} < w_i < \frac{B}{2}$ for all i. The question is whether there exists a partition $S_1, \ldots, S_n$ such that $w(S_j) = B$ for all j, where $w(S_j) = \sum_{i \in S_j} w_i$.

Given an instance of 3-PARTITION, we construct the following instance for the unrestricted DELIVERY problem. There are $n + 1$ terminals $v_0, \ldots, v_n$ and connections (v_i, v_{i-1}) for $i = 1, \ldots, n$. The cost of using the connection from terminal v_1 to terminal v_0 is equal to one, while all other connections have a cost of zero. The transit time for each connection is equal to one. For each weight w_i there

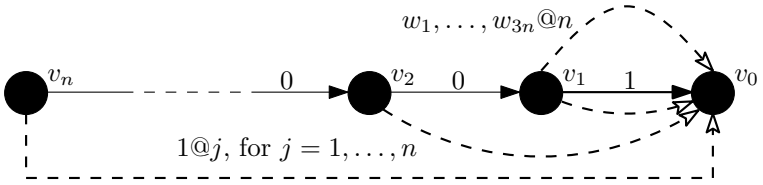

Figure 3: DELIVERY instance with commodities given as volume @ deadline

is a weight commodity with start terminal v_1, end terminal v_0, volume w_i, and deadline for delivery n. Furthermore, for each $j = 1, \ldots, n$ there is a dummy commodity with start terminal v_j, end terminal v_0, volume 1, and deadline j. The capacity of the trucks is equal to $B + 1$. See Figure 3, where the connections are depicted by solid lines, and the commodities by dashed lines.

Consider a solution to this instance of DELIVERY. Because of the tight deadlines for the dummy commodities, these commodities have to be sent immediately. Hence, there will be at least one truck on connection (v_1, v_0) at times $0, \ldots, n-1$, each with an unused capacity of B. Since only this connection contains non-zero cost, this implies that the minimum possible cost for this instance is n. This cost can only be obtained if the weight commodities can be split into n sets, each consisting of three commodities, with the total volume of each set equal to B. Hence, if a solution of cost n exists, there also exists a solution to 3-PARTITION. It is also easy to see that if a solution to 3-PARTITION exists, there is also a solution for DELIVERY with cost n. □

Finally, we consider the case where the DELIVERY problem is restricted to a single commodity.

Theorem 2.5. *The* DELIVERY *problem restricted to a single commodity is NP-hard.*

Proof. The DELIVERY problem restricted to a single commodity is equivalent to the SHORTEST WEIGHT-CONSTRAINED PATH problem [38], with s, t, and W equal to the start terminal, end terminal, and deadline for the commodity, respectively, and the length and weight of connection (i, j) set to cost d_{ij} and transit time τ_{ij}, respectively. □

2.3 LP-BASED ALGORITHMS

Having settled the complexity of DELIVERY, in this section we propose several algorithms that obtain good solutions in a reasonable amount of time. Thereto, we first formulate the problem at hand as a mixed inter program (MIP), both for a restricted and unrestricted set of allowed routes, and then describe two LP-based rounding algorithms.

2.3.1 Restricted MIP Formulation

The MIP formulation for the DELIVERY problem with a restricted set of allowed routes uses the following variables:

$$y_{ij}^t = \text{number of trucks on connection } (i,j) \text{ at time } t,$$

$$x_{ijr}^t = \begin{cases} 1, & \text{if route } r \text{ uses the connection from terminal } i \text{ to } j \text{ at time } t, \\ 0, & \text{otherwise,} \end{cases}$$

$$z_{ir}^t = \begin{cases} 1, & \text{if route } r \text{ stays at terminal } i \text{ for one time unit at time } t, \\ 0, & \text{otherwise.} \end{cases}$$

Before the mixed integer program can be presented, several sets and parameters need to be defined. Let R denote the set of all allowed routes, i. e., $R = \bigcup_{k \in K} R_k$. The set of commodities with terminal i as start terminal is defined by K_i, while the set of commodities with terminal j as end terminal is described by K^j. Finally, let T denote the set of all time points, i. e., $T = \{0, 1, \ldots, T_{max}\}$. Using these definitions, we arrive at the following formulation.

$$\min \sum_{(i,j)\in A} \sum_{t\in T} d_{ij} y_{ij}^t$$

$$\text{s.t.} \sum_{k\in K} \sum_{r\in R_k} v_k x_{ijr}^t \leqslant c y_{ij}^t, \qquad \forall\, (i,j) \in A, t \in T, \tag{1}$$

$$\sum_{r\in R_k} \sum_{j\in N:(s_k,j)\in r} x_{s_k jr}^0 + z_{s_k r}^0 = 1, \qquad \forall\, k \in K, \tag{2}$$

$$\sum_{j\in N:(s_k,j)\in r} x_{s_k jr}^0 + z_{s_k r}^0 = \sum_{j\in N:(j,t_k)\in r} x_{jt_k r}^{T_{max}-\tau_{jt_k}} + z_{t_k r}^{T_{max}-1}, \qquad \forall\, k \in K, r \in R_k, \tag{3}$$

$$\sum_{j\in N} x_{ijr}^t + z_{ir}^t - \sum_{j\in N} x_{jir}^{t-\tau_{ji}} - z_{ir}^{t-1} = 0, \qquad \forall\, i \in N, k \in K, r \in R_k, t \in T, \tag{4}$$

$$\sum_{j\in N} \sum_{t\in T} y_{ij}^t \geqslant \left\lceil \frac{1}{c} \sum_{k\in K_i} v_k \right\rceil, \qquad \forall\, i \in N, \tag{5}$$

$$\sum_{i\in N} \sum_{t\in T} y_{ij}^t \geqslant \left\lceil \frac{1}{c} \sum_{k\in K^j} v_k \right\rceil, \qquad \forall\, j \in N, \tag{6}$$

$$\sum_{r\in R_k} x_{ijr}^t \leqslant y_{ij}^t, \qquad \forall\, (i,j) \in A, k \in K, t \in T, \tag{7}$$

$$y_{ij}^t \in \mathbb{Z}_+, \qquad \forall\, (i,j) \in A, t \in T, \tag{8}$$

$$x_{ijr}^t \in \{0,1\}, \qquad \forall\, (i,j) \in A, r \in R, t \in T, \tag{9}$$

$$z_{ir}^t \in \{0,1\}, \qquad \forall\, i \in N, r \in R, t \in T. \tag{10}$$

Constraint (1) is a so-called coupling constraint, which enforces that the selected number of trucks on each connection is sufficient to handle the total volume on that connection. Constraint (2) makes sure that exactly one route is selected for each commodity. Constraints (3) and (4) ensure flow conservation. Constraint (3) makes sure that if a route is used to leave the start node of a commodity, then the same route is used to reach the end node of that commodity. Constraint (4) is a standard flow conservation constraint. Note that this constraint does *not* apply to two node and time point combinations, namely the start node of a commodity together with time zero, and the end node of a commodity together

with time T_{max}. Constraints (5) and (6) are so-called cut-set inequalities. Constraint (5) considers the commodities starting from a certain terminal, and states that the total number of trucks leaving a terminal is enough to cover the total volume of all commodities starting from that terminal. Constraint (6) is similar to constraint (5), but now all commodities and trucks that have the terminal as end point are considered. Constraint (7) is a disaggregation of the coupling constraint (1), as described in for instance [44], intended to strengthen the link between the flow (x^t_{ijr}) and design (y^t_{ij}) variables. This type of inequality is known to give a large improvement on the integrality gap in case of binary design variables, at the expense of a high number of extra constraints. However, in our formulation the design variables are integral, but preliminary computational results showed that these inequalities are also useful for our formulation. Therefore, we did not consider a reformulation into $0-1$ design variables, as described in for instance [33].

2.3.2 Unrestricted MIP Formulation

The MIP formulation for the unrestricted DELIVERY problem uses the following variables:

$$y^t_{ij} = \text{number of trucks on connection } (i,j) \text{ at time } t,$$

$$x^t_{ijk} = \begin{cases} 1, & \text{if commodity } k \text{ uses the connection from terminal } i \text{ to } j \text{ at time } t, \\ 0, & \text{otherwise,} \end{cases}$$

$$z^t_{ik} = \begin{cases} 1, & \text{if commodity } k \text{ stays at terminal } i \text{ for one time unit at time } t, \\ 0, & \text{otherwise.} \end{cases}$$

We use the same sets and parameters as defined in Section 2.3.1, with one extra definition. Parameter δ^t_{ik} has value 1 if i is the start terminal of commodity k and the time t equals 0, value -1 if i is the end terminal of commodity k and the time t equals T_{max}, and value 0 otherwise. Using these definitions, we arrive at the following formulation.

$$\min \sum_{(i,j)\in A} \sum_{t\in T} d_{ij} y_{ij}^t$$

$$\text{s.t.} \sum_{k\in K} v_k x_{ijk}^t \leqslant c y_{ij}^t, \qquad \forall\, (i,j) \in A, t \in T, \tag{11}$$

$$\sum_{j\in N} x_{ijk}^t + z_{ik}^t - \sum_{j\in N} x_{jik}^{t-\tau_{ji}} - z_{ik}^{t-1} = \delta_{ik}^t, \qquad \forall\, i \in N, k \in K, t \in T, \tag{12}$$

$$\sum_{j\in N} \sum_{t\in T} y_{ij}^t \geqslant \left\lceil \frac{1}{c} \sum_{k\in K_i} v_k \right\rceil, \qquad \forall\, i \in N, \tag{13}$$

$$\sum_{i\in N} \sum_{t\in T} y_{ij}^t \geqslant \left\lceil \frac{1}{c} \sum_{k\in K^j} v_k \right\rceil, \qquad \forall\, j \in N, \tag{14}$$

$$x_{ijk}^t \leqslant y_{ij}^t, \qquad \forall\, (i,j) \in A, k \in K, t \in T, \tag{15}$$

$$y_{ij}^t \in \mathbb{Z}_+, \qquad \forall\, (i,j) \in A, t \in T, \tag{16}$$

$$x_{ijk}^t \in \{0,1\}, \qquad \forall\, (i,j) \in A, k \in K, t \in T, \tag{17}$$

$$z_{ik}^t \in \{0,1\}, \qquad \forall\, i \in N, k \in K, t \in T. \tag{18}$$

The main difference with the MIP formulation given in Section 2.3.1, is that Constraints (2), (3), and (4) are combined into a single Constraint (12).

2.3.3 Rounding LP Relaxation

Finding an optimal solution for these MIP formulations may take a long time, even for medium sized instances. Therefore, we focus on obtaining feasible solutions within a reasonable amount of time and analyze their worst-case performance guarantee. Hereto, we consider LP-based rounding algorithms. The algorithms that we consider work as follows. First, the LP relaxation of the MIP formulation is solved. Then, a route and time points at which the connections of that route are used, are determined for each commodity, using a rounding scheme for the relaxed variables x_{ijr}^t (x_{ijk}^t). Once the route and time points are determined for all commodities, it is straightforward to determine the volume on each connection, and thus also the required number of trucks on this connection. We consider two rounding schemes, that cannot be applied to the unrestricted MIP formulation (see Section 2.3.2). The reason for this is that it is not clear how

feasible routes can be distilled from this formulation. Hence, we only explain how these schemes can be applied to the MIP formulation for a restricted set of allowed routes (see Section 2.3.1).

The first rounding algorithm that we consider is called *Randomized Rounding*. From constraint (2), we know that the $x^0_{s_k jr}$ and $z^0_{s_k r}$ variables for commodity k sum up to one. Hence, the values for these variables can be seen as a probability distribution. Now, for commodity k select route $\hat{r} \in R_k$ with probability $\mathbf{Pr}\{\hat{r}\} = \sum_{j \in N} x^0_{s_k j \hat{r}} + z^0_{s_k \hat{r}}$. After the route $\hat{r}$ has been selected, the time units at which this route is used can be determined. Let β be a random number, drawn uniformly between 0 and $\mathbf{Pr}\{\hat{r}\}$. Connection (i,j) in route $\hat{r}$ will be used at time $\hat{t}_{ij}$, where $\hat{t}_{ij}$ is such that $\sum_{t<\hat{t}_{ij}} x^t_{ij\hat{r}} < \beta \leqslant \sum_{t\leqslant\hat{t}_{ij}} x^t_{ij\hat{r}}$. Because of the flow balance constraints, the times at which the connections are traversed are consistent with the order in which they appear in the route. It is easily checked that the probability that commodity k uses connection (i,j) of route r at time t is equal to x^t_{ijr}.

Theorem 2.6. *Randomized Rounding is a pseudo-polynomial time* $(|K|+1)$*-approximation algorithm for* DELIVERY.

Proof. Without loss of generality, assume that the capacity is equal to one. This can be achieved by appropriately scaling the volume of the commodities. Let $\alpha^t_{ijk} = \sum_{r\in R_k} x^t_{ijr}$ denote the fraction of commodity k that is transported over connection (i,j) at time t. Using this definition, we can rewrite constraints (1) and (7) as

$$\sum_{k\in K} v_k \alpha^t_{ijk} \leqslant y^t_{ij}, \quad \forall (i,j) \in A,\ \forall t \in T, \tag{19}$$

$$\alpha^t_{ijk} \leqslant y^t_{ij}, \qquad \forall (i,j) \in A,\ \forall k \in K,\ \forall t \in T. \tag{20}$$

Observe that by the definition of Randomized Rounding, the probability that commodity k uses connection (i,j) at time t is equal to α^t_{ijk}. Let TR_S denote the number of trucks needed for commodity set $S \subseteq K$. For TR_S we get that

$$TR_S = \left\lceil \sum_{k \in S} v_k \right\rceil \leqslant \sum_{k \in S} (v_k + 1).$$

Finally, let $\mathbf{Pr}^t_{ij}\{S\}$ denote the probability that precisely commodity set $S \subseteq K$ is randomly selected to traverse connection (i, j) at time t. By definition, we have that

$$\mathbf{Pr}^t_{ij}\{S\} = \prod_{k \in S} \alpha^t_{ijk} \prod_{k \in K \setminus S} \left(1 - \alpha^t_{ijk}\right).$$

Consider an arbitrary connection (i, j) and time point t. Let $\mathbb{E}\left[TR^t_{ij}\right]$ denote the expected number of trucks needed on connection (i, j) at time t. By the definition of expectation, we have:

$$\begin{aligned}
\mathbb{E}\left[TR^t_{ij}\right] &= \sum_{S \subseteq K} \mathbf{Pr}^t_{ij}\{S\} TR_S \\
&= \sum_{S \subseteq K} \prod_{k \in S} \alpha^t_{ijk} \prod_{k \in K \setminus S} \left(1 - \alpha^t_{ijk}\right) TR_S \\
&\leqslant \sum_{S \subseteq K} \prod_{k \in S} \alpha^t_{ijk} \prod_{k \in K \setminus S} \left(1 - \alpha^t_{ijk}\right) \sum_{k \in S} (v_k + 1) \\
&= \sum_{k \in K} \alpha^t_{ijk} (v_k + 1) \\
&= \sum_{k \in K} \alpha^t_{ijk} + \sum_{k \in K} v_k \alpha^t_{ijk} \\
&\leqslant (|K| + 1)\, y_{ijt},
\end{aligned}$$

where the last inequality follows from constraints (19) and (20). Hence, for connection (i, j) at time t we have that the expected number of trucks is at most $|K| + 1$ times the number of trucks needed for the LP relaxation. This implies that the expected cost for connection (i, j) at time t is also at most $|K| + 1$ times the cost in the LP relaxation. As connection (i, j) and time t were chosen arbitrarily, this holds for all connections and all times. Hence, the expected cost of the

solution is at most $|K|+1$ times the cost of the LP relaxation. As the cost of the LP relaxation is also a lower bound for the optimal cost, we have that the expected cost is at most $|K|+1$ times the optimal cost, so that Randomized Rounding is indeed a $(|K|+1)$-approximation algorithm for DELIVERY. □

The second rounding algorithm that we consider is called *Highest Rounding*. This algorithm works as follows. For each commodity k, determine the route $\hat{r}$ that transports the highest fraction of the commodity, i. e., determine the route $\hat{r}$ for which $\sum_{j\in N} x^0_{s_k j \hat{r}} + z^0_{s_k \hat{r}}$ is maximized. After route $\hat{r}$ is selected, it is decomposed into so-called *timed routes*, which for each connection in the route specifies at what time it is used. These timed routes are obtained as follows. For each connection (i,j) in route $\hat{r}$, determine the minimum time t_{ij} for which $x^{t_{ij}}_{ij\hat{r}} > 0$. Because of the flow balance constraints, it is guaranteed that these times t_{ij} constitute a consistent order. The value of this timed route is $\min_{(i,j)\in\hat{r}} x^{t_{ij}}_{ij\hat{r}}$, and this value is subtracted from all $x^{t_{ij}}_{ij\hat{r}}$ variables in the route. This process is repeated until all $x^{t}_{ij\hat{r}}$ variables are equal to zero. Out of all these timed routes, the one with the highest value is selected. Observe that this decomposition is not unique, and that other decompositions can lead to different solutions.

Theorem 2.7. *Highest Rounding is a pseudo-polynomial time* $(|K|+1)R_{max}$ *- approximation algorithm for* DELIVERY, *where* $R_{max} = \max_{k\in K} |R_k|$.

Proof. Assume without loss of generality that the capacity is equal to one. Consider an arbitrary connection (i,j), and let $S^t \subseteq K$ denote the set of commodities that use connection (i,j) at time t for Highest Rounding. By definition we have that $S^t \cap S^{t'} = \emptyset$ for $t \neq t'$. Furthermore, let $S = \cup_{t\in T} S^t$ denote the set of commodities that traverse connection (i,j) for Highest Rounding. Let TR_S denote the number of trucks needed for commodity set S. For TR_S we get that

$$\begin{aligned} TR_S &= \sum_{t\in T} TR_{S^t} = \sum_{t\in T} \left\lceil \sum_{k\in S^t} v_k \right\rceil \leqslant \sum_{t\in T} \sum_{k\in S^t} (v_k + 1) \\ &= \sum_{k\in S} (v_k + 1) = \sum_{k\in S} v_k + |S|. \end{aligned}$$

Let r_k denote the selected route for commodity $k \in S$. From constraint (7) we get that

$$\sum_{t \in T} y_{ij}^t \geqslant \sum_{t \in T} \sum_{r \in R_k} x_{ijr}^t \geqslant \sum_{t \in T} x_{ijr_k}^t \geqslant \frac{1}{|R_k|} \geqslant \frac{1}{R_{max}},$$

where the second to last inequality follows from the fact that among the routes in R_k, route r_k is used for the highest fraction. From constraint (1) we get that

$$\sum_{t \in T} y_{ij}^t \geqslant \sum_{t \in T} \sum_{k \in K} \sum_{r \in R_k} v_k x_{ijr}^t \geqslant \sum_{t \in T} \sum_{k \in S} v_k x_{ijr_k}^t \geqslant \frac{1}{R_{max}} \sum_{k \in S} v_k.$$

Combining these observations, we get that

$$\begin{aligned} TR_S &\leqslant \sum_{k \in S} v_k + |S| \\ &\leqslant R_{max} \sum_{t \in T} y_{ij}^t + |S| \\ &\leqslant R_{max} \sum_{t \in T} y_{ij}^t + |S| R_{max} \sum_{t \in T} y_{ij}^t \\ &= (|S| + 1) R_{max} \sum_{t \in T} y_{ij}^t \\ &\leqslant (|K| + 1) R_{max} \sum_{t \in T} y_{ij}^t, \end{aligned}$$

where the second inequality follows from constraint (1), and the third inequality from constraint (7). Hence, for connection (i, j) we have that the number of trucks is at most $(|K| + 1)R_{max}$ times the number of trucks needed in the LP relaxation. As the connection was chosen arbitrarily, this observation holds for all connections. Hence, the cost obtained by Highest Rounding is at most $(|K| + 1)R_{max}$ times the cost of the LP relaxation, so that Highest Rounding is a $(|K| + 1)R_{max}$-approximation algorithm for DELIVERY. □

2.4 HEURISTICS AND LOCAL SEARCH NEIGHBORHOODS

In the previous section we described two LP-based rounding algorithms. One drawback of these methods is that solving the LP relaxation for large instances may take a long time. Therefore, we describe constructive heuristics, that construct a feasible solution from scratch, and improvement methods, that try to improve a given feasible solution.

2.4.1 Constructive Heuristics

We consider two constructive heuristics, called *Independent Shortest Path* and *Sequential Shortest Path*, that both work according to the same principle. First, the commodities are sorted in a specified order. Next, the commodities are considered in this order, and for each commodity the route that minimizes the cost, according to some measure, is selected.

For ordering the commodities we consider three options, called *Lexicographic*, *Volume Ascending*, and *Volume Descending*. For Lexicographic the commodities are sorted first on the id of the start terminal and then the id of the end terminal. For Volume Ascending and Volume Descending the commodities are sorted on volume, in ascending and descending order, respectively. In case of ties, the commodities are sorted in Lexicographic order.

For Independent Shortest Path the route that minimizes the cost for the commodity is determined. As the cost of a connection does not depend on the time that it is used, all connections in the selected route can be used as early as possible. Hence, the decision as to which route to take solely depends on the sum of costs for each connection in the route.

For Sequential Shortest Path the route and time points that minimizes the *additional* cost, given the choices for the previously considered commodities, is selected.

Note that for Independent Shortest Path the order in which the commodities are considered does not influence the solution. Hence, in case Independent Shortest Path is applied, the commodities are always considered in Lexicographic order. For Sequential Shortest Path the order could influence the solution.

2.4.2 Local Search Methods

All improvement methods described in this section are based on the local search form *iterative improvement*. Hence, only moves to strictly better solutions are considered. For selecting a neighboring solution, we consider two schemes, called *first* and *best* improvement.

The only difference between the local search methods is in the definition of the neighborhoods. Thereto, we only describe how the neighborhood of the current solution is obtained. The three neighborhoods that we consider are called *Commodity Neighborhood*, *Connection Neighborhood*, and *Combined Neighborhood*.

The Commodity Neighborhood is obtained as follows. Consider a commodity k and remove it from the current solution, to obtain a partial solution. Then, for commodity k the route and time points that minimize the total cost, given the partial solution, is determined. Note that this step is similar to determining the route and time points for the Sequential Shortest Path heuristic. This procedure is repeated for all commodities.

For the Connection Neighborhood, consider a connection (i, j) and a time point t. Let K_{ij}^t denote the set of commodities that traverse connection (i, j) at time t. From the current solution, remove commodities $k \in K_{ij}^t$ to obtain a partial solution. Next, consider these removed commodities in Volume Descending order, and determine the route and time points that minimize the additional cost, similarly as for Sequential Shortest Path. This procedure is repeated for all connections and time points.

As the name already indicates, the Combined Neighborhood is a combination of the Commodity Neighborhood and the Connection Neighborhood. A solution is in the Combined Neighborhood if it is an element of either the Commodity Neighborhood or the Connection Neighborhood.

As the local search methods that we described only allow improving steps, it is possible that the procedure gets stuck in a local optimum after only a short amount of time, exploring only a small amount of the solution space. One option to circumvent this issue is to *reshuffle* the solution once a local optimum is found. The current solution is reshuffled by randomly selecting a specified number of commodities, and then randomly selecting a route (including time points) for

each of these commodities. After the new start solution is determined, a local search procedure is selected at random. This procedure, called *Reshuffle Random*, is repeated until a certain time limit is reached. Observe that the new start solution is obtained from the *current* local optimum, and not from the best solution found so far. The idea behind this decision is to explore a larger part of the solution space.

2.5 EXPERIMENTAL RESULTS

We evaluate the algorithms described in Sections 2.3 and 2.4 on randomly generated instances, that are based on problems obtained from TSPLIB [64]. We first explain how the instances are constructed, followed by an overview of the results.

2.5.1 Setup

The algorithms are implemented in C++ and CPLEX 12.2 is used to solve the linear programs. The instances are solved on a machine with 2 Intel Xeon CPUs running at 2.53 Ghz with 8 GB RAM. The instances are based on eight problems from TSPLIB, namely gr24, fri26, hk48, berlin52, eil76, pr76, kroA100, and rd100. These problems already contain information on the set of terminals and distances between these terminals. Observe that the number of terminals for each problem is equal to the number in its name. To obtain instances for DELIVERY, transit times, truck capacity, and commodities with their allowed routes still need to be added.

We chose the transit time of a connection to be proportional to the distance, i. e., $\tau_{ij} = \lceil 5 \times d_{ij}/d_{max} \rceil$ for connection (i, j). Hence, the transit time is an integer number between 1 and 5. The capacity of the trucks is set to 1. For each problem from TSPLIB, we create three types of instances for DELIVERY, depending on the number of commodities. For the *low*-type instance, the number of commodities is 10% of the total number of possible commodities, where the total number of possible commodities is $|N|(|N|-1)$. For the *medium*- and *high*-type instances, the number of commodities is 25% and 50% of the possible number of commodities, respectively. For each commodity, the start and end terminal are selected randomly. From instances seen in practice, we have noticed that there

are many commodities that have a rather small volume, and only a few with a large volume. By scaling, we may assume that the volume of each commodity is at most 1. We have chosen the volumes to be in the set $\{1 \cdot 10^{-5}, 2 \cdot 10^{-5}, \ldots, 1\}$, which are randomly selected. To ensure that there will be more small volumes than large volumes, we decided to choose a random $x \in (0, 1]$ from a distribution with CDF $F(x) = x^{12}$, and then round it to the nearest integer multiple of 10^{-5}. The deadline for each commodity is chosen to be twice the length, with respect to the transit times, of the shortest route from the start to the end terminal.

For the set of specified routes, we consider three options. For the first option we allow all possible routes, called the *Unrestricted* setting. For the second and third option, we specify special terminals, called *hubs*, that will act as collection and distribution terminals. For each problem, three hubs are selected in the following way. Consider a subset of three terminals, and for all terminals determine the distance to the nearest of these three terminals. The total distance for this subset is the sum of distances over all terminals. The subset of terminals with the lowest total distance is selected as set of hubs. For the second option, called *All Hubs*, the direct route from the start terminal to the end terminal is allowed, as well as all routes with one, two, or three hubs as intermediate terminals. This results in at most 16 routes per commodity. For the third option, called *Nearest Hub*, at most 4 routes are allowed: directly from start to end terminal, using the nearest hub of the start terminal as intermediate terminal, using the nearest hub of the end terminal as intermediate terminal, and using respectively the nearest hub of the start and end terminal as intermediate terminals. Observe that multiple copies of the same route, routes that contain multiple copies of a terminal, and routes that do not allow the timely delivery of the commodity, are discarded.

Concluding, we have eight TSPLIB problems on which the instances are based, three options for the number of commodities, and three options for the allowed routes. This results in 72 unique groups of settings. For each combination of TSPLIB problem and number of commodities, 10 random instances are generated, resulting in 240 instances in total.

2.5.2 Results

For all 72 combinations of the problem instance, number of commodities, and set of allowed routes, we aggregate the results of the 10 randomly generated instances. First, we compare the results for the heuristics, to determine the starting point for the local search. Then, we compare the local search results, followed by an overview of the results for the MIP and the LP-based rounding algorithms. Finally, we compare all these methods to determine which one performs best. An overview of the results for the heuristics, local search neighborhoods, and LP-based algorithms can be found in Appendix 2.A.

Within the forthcoming sections we also summarize the results for ease of reference. To be able to compare the results for all different instances, we determine for each of the 240 instances the ratio between the obtained solutions for all three options for the allowed routes and the best found feasible solution. Each overview consists of four columns, representing the results over all instances, and the results split for the Unrestricted, All Hubs, and Nearest Hub settings. For each entry the average ratio is given, followed by the average run time (in seconds) between brackets.

2.5.2.1 *Heuristics*

As already stated in Section 2.4.1, the order in which the commodities are considered does not influence the solution for Independent Shortest Path, but it does for Sequential Shortest Path. The expectation is that Independent Shortest Path has worse results compared to Sequential Shortest Path. The reason for this is that Independent Shortest Path does not take the selected routes for the other commodities into account, and is thus not looking for possibilities to combine commodities. Among the three orderings for Sequential Shortest Path, the expected result is that Volume Descending has on average the lowest cost. For this ordering the bigger commodities are considered first, which could be difficult to combine, followed by the smaller commodities, which hopefully fit in the remaining capacity, so that the smaller commodities could be routed at a small, or even no, additional cost. With respect to the running time the expectation is that it will be very low, with no big differences between Independent Shortest Path and Sequential Shortest Path, and that the running time will be higher with

more commodities and more possible routes. For a summary of the results for the heuristics, see Table 1.

	All Instances		Unrestricted		All Hubs		Nearest Hub	
Independent	4.87	(0.63)	4.76	(1.71)	4.92	(0.09)	4.92	(0.09)
Lexicographic	2.41	(0.38)	1.37	(0.95)	2.82	(0.09)	3.05	(0.09)
Volume Asc	2.64	(0.37)	1.47	(0.93)	3.12	(0.10)	3.34	(0.09)
Volume Desc	2.63	(0.39)	1.58	(0.98)	3.04	(0.10)	3.27	(0.09)

Table 1: Summary of the results for the heuristics. The first row depicts the results for Independent Shortest Path. The last three rows depict the results for the three Sequential Shortest Path orderings Lexicographic, Volume Ascending, and Volume Descending.

Among the heuristics, Independent Shortest Path has the highest cost for all 72 groups. Over all groups, the cost obtained by Independent Shortest Path is on average 2.1 times higher, and maximally 5.1 times higher, as for the worst ordering for Sequential Shortest Path. In general we have that the higher the number of commodities and the higher the number of allowed routes, the worse Independent Shortest Path performs compared to Sequential Shortest Path. An explanation for this is that with more commodities and more possible routes, Sequential Shortest Path has more options for combining commodities.

All three Sequential Shortest Path orderings clearly outperform Independent Shortest Path. A surprising result is that among the three orderings, Lexicographic performs best. Lexicographic gives the best result in 66 out of 72 groups.

The results for the running time are in general as expected. The average running time over all instances and all four settings is 0.44 seconds. By far the highest contribution stem from the Unrestricted instances. The only surprising result is the difference in running time between Independent Shortest Path and Sequential Shortest Path for the Unrestricted instances. On average the running time for Independent Shortest Path is 52.7% higher than the running time for Sequential Shortest Path, and in the worst case even 99.5%. As the maximum running

time is only 9.32 seconds, we have not extensively analyzed the reason for this difference.

Concluding, we have that Sequential Shortest Path with ordering Lexicographic performs best, with 0.38 and 5.24 seconds as average and maximum running time, respectively.

2.5.2.2 *Local Search Neighborhoods*

In Section 2.4.2 we described three neighborhoods and two schemes for selecting a neighboring solution. This results in six different combinations, which are all run using Sequential Shortest Path with ordering Lexicographic as start point, the best performing heuristic (see Section 2.5.2.1).

	All Instances		Unrestricted		All Hubs		Nearest Hub	
first	1.52	(6.25)	1.15	(16.42)	1.65	(1.19)	1.76	(1.14)
best	1.52	(342.29)	1.15	(805.20)	1.65	(109.91)	1.76	(111.75)

Table 2: Summary of results for the local search neighborhoods - first improvement vs best improvement. The first row depicts the results for the first improvement schemes, the second row the results for the best improvement schemes.

First, we compare the first improvement schemes with the best improvement schemes. For a summary of the results, see Table 2. The expectation is that the best improvement schemes will perform better on solution quality, as always the best neighboring solution is selected, while the first improvement schemes are expected to perform better on running time, as not necessarily all neighbors need to be considered before selecting a neighboring solution. As it turns out, the difference between these schemes is very small in terms of solution quality. The cost for the first improvement schemes is on average 0.02% higher than the cost for the best improvement schemes, with the difference in cost ranging between -2.59% and 3.19%. If we now consider the running time, we observe a clear difference. The maximum running time for first improvement is 208.11 seconds, while for best improvement the maximum running time is over 3 hours. Hence, we have that best improvement slightly performs better on solution qual-

ity, while first improvement clearly performs better on running time. Thereto, from now on we only compare the first improvement schemes.

	All Instances		Unrestricted		All Hubs		Nearest Hub	
Comm	1.62	(1.28)	1.26	(3.00)	1.75	(0.43)	1.85	(0.41)
Conn	1.47	(9.34)	1.10	(23.78)	1.60	(2.14)	1.71	(2.11)
Comb	1.47	(8.12)	1.10	(22.48)	1.60	(0.99)	1.71	(0.90)
ResRan	1.24	(300.00)	1.00	(300.00)	1.33	(300.00)	1.40	(300.00)

Table 3: Summary of the results for the local search neighborhoods - first improvement. The first three rows depict the results for respectively Commodity Neighborhood, Connection Neighborhood, and Combined Neighborhood. The last row depicts the results for Reshuffle Random.

For a summary of the results for the three neighborhoods for first improvement, see Table 3. If we compare the three neighborhoods, we see that Commodity Neighborhood clearly has the worst results. The difference between Connection Neighborhood and Combined Neighborhood is a lot smaller. On average, the cost for Connection Neighborhood is only 0.04% lower than the cost for Combined Neighborhood, while the maximum difference between the two neighborhoods is 1.65%. Also in terms of running time there is little difference, with Connection Neighborhood having a slightly longer running time. If we compare the results of the two neighborhoods with the results obtained by Sequential Shortest Path with ordering Lexicographic, we see that on average the two neighborhoods obtain an improvement of 34.73%.

As there is almost no difference between Connection Neighborhood and Combined Neighborhood, we decided to use the following settings for Reshuffle Random: the time limit is 300 seconds, and the initial solution is obtained by Sequential Shortest Path with ordering Lexicographic. At the start of each iteration one of the three first improvement schemes is selected at random, and at the end of the iteration 25% of the commodities are reshuffled, if the time limit is not yet reached.

If we compare Reshuffle Random with the Connection Neighborhood and the Combined Neighborhood, we observe a clear improvement obtained by Reshuffle Random. Over all groups, Reshuffle Random outperforms the two neighborhoods by 13.83% on average. In general, we have that the lower the number of commodities, and the lower the number of possible routes, the better the (relative) performance of Reshuffle Random is. This can be explained as follows: with a lower number of commodities and/or a lower number of possible routes, the running time of all three neighborhoods decreases. This implies that more iterations can be performed in the 300 seconds allotted to Reshuffle Random.

Concluding, we have that first improvement is preferred over best improvement because of a substantial lower running time. Of the three proposed neighborhoods, Commodity Neighborhood performs worst, while there is almost no difference between Connection Neighborhood and Combined Neighborhood. On average these neighborhoods obtain an improvement of 34.73%. By running Reshuffle Random, an additional improvement of 13.83% can be obtained. Hence, on average an improvement of 43.41% is obtained by Reshuffle Random compared to Sequential Shortest Path with ordering Lexicographic.

2.5.2.3 *LP-based Algorithms*

For solving the MIPs, CPLEX is given a time limit of 900 seconds, with intermediate results obtained after 300 and 600 seconds. The LP relaxation is solved via cut generation. The rounding procedure for Randomized Rounding is applied 10,000 times. As the rounding procedure for Highest Rounding is deterministic, it only needs to be applied once. For a summary of the results, see Table 4.

Consider the MIP results. Observe that for the Unrestricted instances, results are only obtained for instances that are based on gr24 and fri26. The reason for this is that the model is too large for the other instances, i. e., the maximum number of constraints that CPLEX can handle is exceeded. Out of the 540 remaining instances, 178 are solved to optimality within 900 seconds. The optimal solution is obtained within the first 300 seconds for 160 instances, within the second 300 seconds for 10 instances, and within the last 300 seconds for 8 instances. Hence, for the majority of the instances we have that the optimal solution is either obtained within 300 seconds, or not obtained within 900 seconds. If we

	All Instances		Unrestricted		All Hubs		Nearest Hub	
MIP300	1.91	(228.29)	3.19	(279.24)	1.88	(230.99)	1.62	(212.84)
MIP600	1.89	(440.56)	3.09	(533.99)	1.87	(450.51)	1.61	(407.26)
MIP900	1.88	(646.26)	3.01	(779.88)	1.87	(662.27)	1.60	(596.78)
RanRou	1.45	(29.09)		—	1.44	(29.80)	1.47	(28.37)
HiRou	1.45	(0.19)		—	1.42	(0.21)	1.47	(0.17)
LP	1.23	(85.13)		—	1.19	(136.47)	1.27	(33.80)

Table 4: Summary of the results for the LP-based algorithms. The first three rows depicts the results obtained for the MIP formulation after respectively 300, 600, and 900 seconds. The next two rows depict the results for Randomized Rounding and Highest Rounding. The last row depicts the results obtained for the LP relaxation.

consider the Unrestricted, All Hubs, and Nearest Hub instances separately, then 12, 75 and 91 instances are solved to optimality, respectively. There are also some surprising results. By definition, we have that the set of routes that are allowed for Nearest Hub is a subset of the routes allowed for All Hubs, which again is a subset of the routes allowed for Unrestricted. Hence, we have that an optimal solution for an Unrestricted instance is at least as good as an optimal solution for an All Hubs instance, which in turn is at least as good as an optimal solution for a Nearest Hub instance. If we now consider the six groups for which all results can be obtained, we observe that for only two groups Unrestricted performs best, while for the other four Unrestricted performs worst. If we compare Nearest Hub with All Hubs, we see that for 8 out of 24 groups the solution obtained by Nearest Hub is strictly better than the solution obtained by All Hubs. This warrants the choice to restrict the set of allowed routes, as good solutions might be obtained in a shorter time, i. e., better solutions might be obtained in the same amount of time. Also surprising is that there are two instances that are solved to optimality for All Hubs, but not for Nearest Hub.

If we want to compare the LP-based rounding algorithms, we first need to solve the LP relaxation. As Randomized Rounding and Highest Rounding cannot be executed for the Unrestricted instances, we do not solve the LP relaxation for these instances. The solution for the LP relaxation is obtained with the help

of cut generation. The vast majority of the constraints are of type (7) (see Section 2.3.1). Adding all these constraints, of which only a small fraction will be binding, would greatly increase the running time. Therefore, the LP relaxation is iteratively solved, where after each iteration all violated constraints of type (7) are added. This process is continued until there are no violated constraints. The average running time needed to solve the LP relaxation is 85.13 seconds, and the maximum time is 720.83 seconds. The average running time for the Nearest Hub instances is substantially lower than for the All Hubs instances.

On average, the cost obtained by Randomized Rounding is 0.57% higher than the cost obtained by Highest Rounding, with a difference in cost of at most 5.28%. We observe that Randomized Rounding scores relatively better than Highest Rounding in groups with a low number of commodities or a low number of allowed routes. A possible reason is that for a higher number of commodities of allowed routes, the probability that the randomly determined solution uses non-combining routes is higher, i. e., there is a higher probability of a bad solution.

If we compare the running time, we see that the running time for Highest Rounding is considerably lower than for Randomized Rounding. This follows from the fact that rounding is performed 10,000 times for Randomized Rounding and only once for Highest Rounding. If we compare the running time of Randomized Rounding for the All Hubs and Nearest Hub instances, we see almost no difference, which implies that the time needed does not heavily depend on the number of allowed routes.

Concluding, we have that the difference in performance between Randomized Rounding and Highest Rounding is very small. There is a difference in running time, although the average running time for Randomized Rounding is only 29.09 seconds, which is still very good. Hence, we keep both methods in consideration in a comparison with the MIP results.

Remember that a time limit of 900 seconds is used for solving the MIP, so that the results obtained by the MIP are not necessarily optimal. Because of the low difference in solution value and low running time for the rounding algorithms, we compare the MIP results with the best performing rounding algorithm. Over all groups, we have that the cost for the MIP solution is on average 21.96% higher. For the All Hubs and Nearest Hub groups, the difference in favor of the round-

ing algorithms is 33.37% and 10.54%, respectively. But if we now consider the number of groups for which each method performs best, we get a very different result. Out of 48 groups, MIP gives the better result in 39 groups, so that the rounding algorithms perform better in only 9 groups. If we now check for which groups MIP scores worse, we see that this occurs for the groups with the most commodities, i. e., for the groups with the largest instances. It appears that for these instances, CPLEX is not able to find a reasonable solution within 900 seconds. If we now only consider the groups for which MIP performs better, we see that the cost for MIP is on average 6.85% lower, while for the groups for which MIP performs worse, we observe that the cost for MIP is on average 146.78% higher. Hence, MIP performs better on the smaller instances, but it has its limitations on the larger instances.

2.5.2.4 *Overall comparison*

To determine which method performs best, we compare the results obtained for MIP, Randomized Rounding, Highest Rounding, and Reshuffle Random. For a summary of these results, see Table 5. Reshuffle Random has one distinct advantage, as it is the only method that can be applied to all instances. Also if we compare it with the two LP-based rounding schemes, we see a clear advantage. On average, Reshuffle Random performs 5.75% better than the rounding schemes, and for all 48 groups for which the rounding schemes could be applied, Reshuffle Random scores better. Hence, between these two methods, Reshuffle Random is preferred. If we compare the MIP results with Reshuffle Random, wee see the same dichotomy as before. On average the cost for MIP is 48.73% higher, but for 38 out of 54 groups MIP obtains the better results. On the groups were MIP performs better, the average cost of MIP is 2.77% lower. Similarly, on groups were Reshuffle Random performs better, the average cost of Reshuffle Random is 171.05% lower. Concluding, we have that Reshuffle Random is our method of choice. It can be applied to all instances, and we have direct control over its running time. It also produces good quality solutions for all instances. This is in contrast with the MIP solution, which is quite often the best solution, but for a decent amount of instances the solution quality is very bad.

	All Instances		Unrestricted		All Hubs		Nearest Hub	
MIP900	1.88	(646.26)	3.01	(779.88)	1.87	(662.27)	1.60	(596.78)
RanRou	1.45	(29.09)		—	1.44	(29.80)	1.47	(28.37)
HiRou	1.45	(0.19)		—	1.42	(0.21)	1.47	(0.17)
ResRan	1.24	(300.00)	1.00	(300.00)	1.33	(300.00)	1.40	(300.00)

Table 5: Comparison of the results. The first row depicts the results of the MIP formulation after 900 seconds. The second and third row depict the results for Randomized Rounding and Highest Rounding, respectively. The last row depicts the results for Reshuffle Random.

2.6 CONCLUSION

In this chapter we considered the express delivery problem. Given is a road network, with a distance and travel time for each connection, and a set of commodities. A commodity is defined by a start and end point, a volume, and a deadline for delivery. The goal is to find a minimum cost solution, respecting all deadlines. The costs only stem from operating trucks, each with a fixed capacity. Several settings for this problem are considered, differing in the number of allowed routes for the commodities. We show that this problem is not only NP-hard, but also hard to approximate within a factor logarithmic in the number of commodities. We furthermore evaluate several solution methods on 240 randomly generated instances. Among the methods are LP-based algorithms, heuristics, and local search. Although the results obtained by the MIP formulation are most often the best, we recommend using Reshuffle Random, which applies local search. Its advantage is that it gives feasible results for all instances (the MIP formulation does not), which are generally of a good quality.

Based on the obtained results, we see two directions for further research. The first direction concerns the MIP formulation. One option would be to check if all constraints are necessary. Consider for instance the flow balance constraints. If there is a pair of terminal and time, such that for a commodity it is not possible to reach the terminal by that time, then adding the flow balance constraint for this pair of terminal and time is redundant. By removing these redundant

constraints, the size of the model can be decreased, which should increase the speed with which the model is solved. A further option would be to check if an improvement can be obtained by changing some settings in CPLEX. A final suggestion would be to investigate whether a branch-and-price, or possibly also a branch-and-price-and-cut scheme would help. As we already observed, reducing the number of allowed routes, i. e., going from setting All Hubs to Nearest Hub, sometimes results in a better solution in case of a time limit. We hope that starting with a (very) limited set of routes, and only adding routes if needed, will increase the solution speed. The second direction concerns the application of local search. Reshuffle Random performs very well, but its start solution is obtained by heuristic Sequential Shortest Path with ordering Lexicographic. Compared with the LP-based rounding schemes, this heuristic performs quite bad, i. e., the initial solution with which the local search is started is quite bad. One option would be to start with a solution obtained from one of the rounding schemes. Starting from a better solution will hopefully make it easier to find better local optima. Another adaptation could be to use for instance simulated annealing. For the current local search methods only improving steps are allowed, but allowing non-improving steps might also help. A final suggestion concerns the calibration of Reshuffle Random. Currently, always 25% of the commodities are selected and given a new route. A different value, or possibly even a varying value, might also improve the results.

2.A OVERVIEW OF RESULTS

This section contains an overview of the results for the heuristics, local search neighborhoods, and LP-based algorithms.

First, the results for the heuristics are considered. These results are split into the Unrestricted case in Table 6, the All Hubs case in Table 7, and the Nearest Hub case in Table 8. The column named Independent contains the results for Independent Shortest Path, and the columns Lexicographic, Volume Asc, and Volume Desc contain the results for Sequential Shortest Path with ordering Lexicographic, Volume Ascending, and Volume Descending, respectively.

Second, the results for the local search neighborhoods are considered. For all instances the start solution is obtained by running Sequential Shortest Path with

ordering Lexicographic. The results are split into the Unrestricted case in Table 9, the All Hubs case in Table 10, and the Nearest Hub case in Table 11. The two columns for Commodity, Connection, and Combined contain the results for the first and best improvement scheme for the respective neighborhood. The final column contains the results for the Reshuffle Random neighborhood.

Third, the results for the LP-based algorithms are considered. These results are split into the Unrestricted case in Table 12, the All Hubs case in Table 13, and the Nearest Hub case in Table 14. The columns named MIP300, MIP600 and MIP900 contain the MIP results obtained by CPLEX after 300, 600, and 900 seconds, respectively. Columns RR and HR contain the results for Randomized Rounding and Highest Rounding, respectively, while the values obtained for the LP relaxation are depicted in column LP.

Group	Independent	Lexicographic	Volume Asc	Volume Desc
gr24_lo	7,395	4,876	5,010	5,001
gr24_md	16,901	8,362	8,667	8,561
gr24_hi	29,665	11,772	12,036	12,416
fri26_lo	6,830	4,222	4,614	4,562
fri26_md	16,699	6,478	7,746	7,869
fri26_hi	33,032	9,304	10,835	11,415
hk48_lo	231,093	99,229	113,519	116,219
hk48_md	568,298	162,536	172,875	196,761
hk48_hi	1,123,780	257,562	271,682	303,053
berlin52_lo	153,060	64,060	70,040	71,013
berlin52_md	377,536	108,986	118,224	128,315
berlin52_hi	748,760	166,121	179,436	192,680
eil76_lo	18,263	6,302	6,915	7,321
eil76_md	44,843	10,879	11,445	12,772
eil76_hi	86,668	17,796	18,728	20,216
pr76_lo	4,314,227	1,445,576	1,599,000	1,791,344
pr76_md	10,675,125	2,525,956	2,667,909	3,109,669
pr76_hi	21,387,189	4,082,769	4,265,352	4,791,658
kroA100_lo	1,700,593	468,589	511,341	561,245
kroA100_md	4,209,366	857,307	901,207	1,009,398
kroA100_hi	8,342,986	1,463,055	1,543,680	1,634,866
rd100_lo	547,874	158,684	171,013	190,456
rd100_md	1,356,729	286,121	296,964	334,581
rd100_hi	2,680,151	484,618	508,665	538,324

Table 6: Results for the heuristics for setting Unrestricted

Group	Independent	Lexicographic	Volume Asc	Volume Desc
gr24_lo	8,217	7,447	7,654	7,620
gr24_md	20,240	14,034	15,624	15,780
gr24_hi	38,789	19,556	23,824	22,679
fri26_lo	6,894	6,218	6,268	6,391
fri26_md	16,867	12,298	12,961	13,438
fri26_hi	33,623	18,363	18,697	19,448
hk48_lo	232,166	203,876	210,690	198,481
hk48_md	576,303	362,973	412,835	375,377
hk48_hi	1,153,462	513,604	576,541	544,218
berlin52_lo	153,484	136,684	143,890	142,416
berlin52_md	380,792	236,518	268,528	251,084
berlin52_hi	761,244	329,060	355,380	344,855
eil76_lo	18,771	14,802	16,724	16,058
eil76_md	47,222	23,011	28,202	27,323
eil76_hi	94,171	34,271	37,758	37,592
pr76_lo	4,319,552	3,457,116	3,879,582	3,806,733
pr76_md	10,698,942	4,994,298	6,582,290	6,284,419
pr76_hi	21,465,895	6,704,576	8,402,245	8,554,607
kroA100_lo	1,705,353	1,307,396	1,450,545	1,394,335
kroA100_md	4,242,292	1,946,403	2,198,391	2,212,316
kroA100_hi	8,470,204	2,861,402	3,098,130	3,009,879
rd100_lo	550,712	450,900	470,274	458,384
rd100_md	1,371,379	696,315	775,041	750,332
rd100_hi	2,740,546	1,100,535	1,037,318	1,044,753

Table 7: Results for the heuristics for setting All Hubs

Group	Independent	Lexicographic	Volume Asc	Volume Desc
gr24_lo	8,217	7,533	7,776	7,772
gr24_md	20,240	14,846	16,364	16,310
gr24_hi	38,789	20,153	24,996	23,608
fri26_lo	6,894	6,410	6,392	6,615
fri26_md	16,879	13,530	14,546	14,732
fri26_hi	33,714	21,042	23,220	23,702
hk48_lo	232,166	217,303	217,825	205,585
hk48_md	576,303	412,587	451,206	416,582
hk48_hi	1,153,462	581,434	651,974	599,402
berlin52_lo	153,484	142,745	146,564	145,332
berlin52_md	380,792	267,159	291,134	284,082
berlin52_hi	761,244	352,993	389,827	386,715
eil76_lo	18,771	15,963	17,195	16,486
eil76_md	47,222	25,121	29,974	29,437
eil76_hi	94,171	36,227	41,066	39,709
pr76_lo	4,319,552	3,616,836	3,997,982	3,943,922
pr76_md	10,698,942	5,517,791	7,152,961	6,808,279
pr76_hi	21,465,895	7,453,930	9,134,540	9,192,309
kroA100_lo	1,705,556	1,424,627	1,524,021	1,515,094
kroA100_md	4,243,270	2,284,868	2,435,176	2,475,904
kroA100_hi	8,473,307	3,344,422	3,396,352	3,299,557
rd100_lo	550,712	466,080	488,711	479,962
rd100_md	1,371,379	742,247	828,227	796,304
rd100_hi	2,740,546	1,116,233	1,094,733	1,092,239

Table 8: Results for the heuristics for setting Nearest Hub

Group	Commodity		Connection		Combined		Reshuffle
	first	best	first	best	first	best	Random
gr24_lo	4,406	4,408	3,941	3,911	3,928	3,909	3,317
gr24_md	7,397	7,376	6,324	6,300	6,249	6,304	5,258
gr24_hi	10,884	10,820	9,082	9,211	9,131	9,228	8,002
fri26_lo	3,628	3,632	3,095	3,137	3,130	3,143	2,620
fri26_md	5,757	5,741	4,969	5,031	4,962	5,031	4,216
fri26_hi	8,496	8,485	7,458	7,491	7,565	7,496	6,644
hk48_lo	89,144	89,290	73,550	74,346	74,427	74,364	63,720
hk48_md	150,557	150,127	129,605	130,170	129,024	130,124	116,176
hk48_hi	244,262	243,431	214,592	215,968	215,112	215,867	201,709
berlin52_lo	57,145	56,996	49,308	48,838	49,076	48,887	42,405
berlin52_md	100,871	100,572	86,639	85,895	86,289	85,677	77,548
berlin52_hi	156,107	155,632	136,954	137,695	137,745	137,974	129,970
eil76_lo	5,788	5,779	4,915	4,886	4,948	4,920	4,371
eil76_md	10,321	10,287	9,049	9,072	9,109	9,071	8,591
eil76_hi	17,161	17,115	15,700	15,758	15,741	15,822	15,348
pr76_lo	1,317,617	1,310,991	1,127,155	1,130,694	1,133,574	1,130,874	1,007,534
pr76_md	2,369,715	2,355,058	2,070,837	2,077,684	2,071,751	2,078,582	1,969,171
pr76_hi	3,907,640	3,893,559	3,545,855	3,558,953	3,565,560	3,560,613	3,486,309
kroA100_lo	431,410	429,991	365,332	366,189	368,848	367,011	343,031
kroA100_md	819,688	816,056	731,687	731,429	734,787	732,686	716,883
kroA100_hi	1,421,519	1,416,678	1,313,536	1,312,204	1,309,764	1,314,124	1,310,105
rd100_lo	146,972	146,147	123,191	124,105	125,254	124,008	113,870
rd100_md	273,375	272,275	242,398	243,154	244,493	243,042	237,021
rd100_hi	470,114	468,687	432,867	433,388	433,550	433,143	431,116

Table 9: Results for the local search neighborhoods for setting Unrestricted

Group	Commodity		Connection		Combined		Reshuffle
	first	best	first	best	first	best	Random
gr24_lo	6,471	6,490	6,435	6,441	6,435	6,441	4,792
gr24_md	10,017	9,995	9,309	9,236	9,249	9,236	7,762
gr24_hi	14,937	14,904	13,584	13,518	13,466	13,502	11,354
fri26_lo	4,974	4,949	4,730	4,719	4,737	4,719	3,470
fri26_md	7,496	7,264	6,872	6,804	6,801	6,804	5,470
fri26_hi	11,212	11,074	9,778	9,891	9,926	9,897	8,255
hk48_lo	127,112	125,993	119,018	119,797	118,639	119,793	88,894
hk48_md	197,376	195,857	178,758	178,175	176,729	177,941	150,424
hk48_hi	297,917	295,199	272,238	271,856	270,748	272,356	241,628
berlin52_lo	78,031	77,362	74,102	73,050	73,344	73,050	54,891
berlin52_md	127,835	127,710	116,498	116,710	116,326	116,479	93,350
berlin52_hi	185,041	184,068	168,583	170,307	169,013	170,116	150,338
eil76_lo	9,212	9,143	8,350	8,356	8,404	8,359	6,798
eil76_md	14,955	14,908	13,482	13,510	13,516	13,490	11,932
eil76_hi	23,310	23,240	21,677	21,777	21,681	21,726	20,081
pr76_lo	1,988,299	1,926,955	1,705,917	1,735,623	1,717,864	1,734,339	1,376,001
pr76_md	3,161,010	3,138,633	2,827,225	2,809,220	2,793,124	2,810,267	2,514,646
pr76_hi	4,975,021	4,950,827	4,639,729	4,614,795	4,616,175	4,616,643	4,318,626
kroA100_lo	678,280	666,019	603,604	602,812	604,481	605,527	480,741
kroA100_md	1,085,542	1,079,862	987,934	996,055	993,762	996,352	887,963
kroA100_hi	1,793,754	1,785,560	1,665,655	1,686,995	1,680,887	1,692,478	1,576,356
rd100_lo	248,338	242,864	222,914	222,930	219,557	223,528	174,405
rd100_md	400,195	398,565	362,152	361,686	362,053	361,803	313,604
rd100_hi	632,357	634,896	592,264	599,472	590,502	601,412	541,619

Table 10: Results for the local search neighborhoods for setting All Hubs

Group	Commodity		Connection		Combined		Reshuffle
	first	best	first	best	first	best	Random
gr24_lo	6,733	6,733	6,715	6,715	6,715	6,715	5,055
gr24_md	10,151	10,217	9,386	9,310	9,424	9,308	8,044
gr24_hi	15,069	15,028	13,382	13,319	13,364	13,325	11,640
fri26_lo	5,674	5,683	5,500	5,499	5,500	5,499	4,009
fri26_md	9,440	9,463	9,067	9,018	9,039	9,035	6,534
fri26_hi	12,963	12,726	11,739	11,834	11,697	11,775	9,538
hk48_lo	145,779	145,029	140,900	139,802	140,862	139,781	94,459
hk48_md	212,467	209,007	188,724	191,844	188,128	192,033	157,626
hk48_hi	313,913	313,164	285,855	287,354	286,260	287,382	249,801
berlin52_lo	86,423	87,524	83,138	84,807	82,878	84,894	57,785
berlin52_md	133,193	132,446	121,110	120,630	121,720	121,012	97,783
berlin52_hi	187,769	186,647	171,280	171,037	170,977	171,618	154,872
eil76_lo	9,551	9,516	8,819	8,896	8,782	8,888	7,050
eil76_md	15,458	15,410	13,903	13,904	13,910	13,870	12,307
eil76_hi	23,893	23,874	22,074	22,178	22,052	22,105	20,395
pr76_lo	1,994,182	1,937,908	1,761,516	1,803,165	1,755,268	1,801,937	1,405,526
pr76_md	3,261,570	3,240,791	2,864,330	2,895,842	2,886,489	2,902,954	2,536,774
pr76_hi	5,055,810	5,037,908	4,696,731	4,679,822	4,678,209	4,704,872	4,321,954
kroA100_lo	732,305	722,008	653,222	656,926	651,456	657,507	516,999
kroA100_md	1,159,778	1,152,236	1,036,394	1,039,237	1,032,409	1,041,306	932,475
kroA100_hi	1,848,603	1,851,213	1,738,436	1,763,034	1,732,924	1,762,713	1,611,220
rd100_lo	250,731	246,220	229,139	224,629	228,614	224,317	178,174
rd100_md	412,322	409,174	366,989	368,799	369,040	369,446	318,046
rd100_hi	646,577	648,009	603,034	609,790	605,465	612,759	542,694

Table 11: Results for the local search neighborhoods for setting Nearest Hub

Group	MIP			RR	HR	LP
	300	600	900			
gr24_lo	3,230	3,227	3,226	-	-	-
gr24_md	10,384	8,761	8,288	-	-	-
gr24_hi	42,518	42,518	42,518	-	-	-
fri26_lo	2,562	2,547	2,543	-	-	-
fri26_md	18,937	17,805	16,308	-	-	-
fri26_hi	35,213	35,213	35,213	-	-	-
hk48_lo	-	-	-	-	-	-
hk48_md	-	-	-	-	-	-
hk48_hi	-	-	-	-	-	-
berlin52_lo	-	-	-	-	-	-
berlin52_md	-	-	-	-	-	-
berlin52_hi	-	-	-	-	-	-
eil76_lo	-	-	-	-	-	-
eil76_md	-	-	-	-	-	-
eil76_hi	-	-	-	-	-	-
pr76_lo	-	-	-	-	-	-
pr76_md	-	-	-	-	-	-
pr76_hi	-	-	-	-	-	-
kroA100_lo	-	-	-	-	-	-
kroA100_md	-	-	-	-	-	-
kroA100_hi	-	-	-	-	-	-
rd100_lo	-	-	-	-	-	-
rd100_md	-	-	-	-	-	-
rd100_hi	-	-	-	-	-	-

Table 12: Results for the LP-based algorithms for setting Unrestricted

Group	MIP			RR	HR	LP
	300	600	900			
gr24_lo	4,790	4,790	4,790	4,886	4,956	4,704
gr24_md	7,724	7,724	7,724	8,203	8,268	7,408
gr24_hi	11,092	11,092	11,092	12,666	12,541	10,322
fri26_lo	3,468	3,468	3,468	3,543	3,583	3,416
fri26_md	5,417	5,417	5,417	5,703	5,749	5,204
fri26_hi	8,035	8,032	8,031	8,754	8,878	7,481
hk48_lo	86,853	86,853	86,853	90,283	90,738	84,384
hk48_md	143,532	143,046	142,911	165,142	161,634	134,271
hk48_hi	251,215	249,436	247,727	281,301	270,105	205,034
berlin52_lo	53,519	53,519	53,519	56,526	57,055	51,724
berlin52_md	89,735	89,156	88,948	101,542	101,232	82,482
berlin52_hi	156,689	155,230	152,459	169,251	167,937	127,623
eil76_lo	6,548	6,545	6,545	7,033	7,004	6,307
eil76_md	12,139	11,892	11,813	13,331	12,793	10,299
eil76_hi	53,367	53,367	53,367	22,373	21,755	16,893
pr76_lo	1,309,274	1,307,065	1,306,621	1,447,862	1,429,435	1,254,248
pr76_md	2,636,509	2,570,616	2,510,510	2,869,369	2,725,426	2,139,949
pr76_hi	10,619,760	10,619,760	10,619,760	4,835,705	4,733,649	3,614,483
kroA100_lo	459,315	457,783	456,850	499,162	496,108	431,770
kroA100_md	2,659,642	2,659,642	2,659,642	989,928	962,571	750,873
kroA100_hi	4,494,092	4,494,092	4,494,092	1,766,881	1,744,123	1,332,894
rd100_lo	164,395	164,058	163,917	181,190	179,488	156,314
rd100_md	882,041	881,829	881,829	351,960	336,798	262,841
rd100_hi	1,576,123	1,576,123	1,576,123	602,030	588,446	450,300

Table 13: Results for the LP-based algorithms for setting All Hubs

Group	MIP			RR	HR	LP
	300	600	900			
gr24_lo	5,054	5,054	5,054	5,096	5,164	4,983
gr24_md	8,008	8,008	8,008	8,230	8,436	7,744
gr24_hi	11,479	11,479	11,479	12,420	12,639	10,702
fri26_lo	4,002	4,002	4,002	4,013	4,047	3,953
fri26_md	6,488	6,488	6,488	6,665	6,780	6,296
fri26_hi	9,323	9,323	9,323	10,086	10,200	8,762
hk48_lo	93,502	93,502	93,502	95,974	97,528	90,866
hk48_md	152,870	152,848	152,788	166,380	168,200	143,787
hk48_hi	238,713	237,360	235,918	273,764	272,508	215,394
berlin52_lo	56,912	56,912	56,912	58,574	59,463	55,070
berlin52_md	93,621	93,616	93,613	103,582	104,868	87,265
berlin52_hi	147,794	147,050	145,556	170,808	170,709	131,812
eil76_lo	6,852	6,852	6,852	7,192	7,245	6,597
eil76_md	11,724	11,622	11,590	13,136	12,864	10,764
eil76_hi	22,494	22,364	21,718	22,480	22,057	17,345
pr76_lo	1,353,249	1,352,980	1,352,857	1,431,039	1,448,802	1,301,727
pr76_md	2,435,805	2,404,144	2,396,183	2,745,061	2,722,645	2,212,271
pr76_hi	5,347,593	5,278,062	5,264,973	4,781,271	4,745,046	3,688,889
kroA100_lo	491,413	491,252	491,218	527,714	527,159	469,621
kroA100_md	977,676	955,926	931,737	1,026,895	996,876	796,088
kroA100_hi	5,050,707	5,050,707	5,050,707	1,800,256	1,770,901	1,362,736
rd100_lo	169,927	169,859	169,828	180,701	180,950	161,776
rd100_md	322,583	311,502	306,979	342,262	336,322	270,317
rd100_hi	1,719,254	1,719,254	1,719,254	597,144	589,703	456,093

Table 14: Results for the LP-based algorithms for setting Nearest Hub

3

CONTAINER PREMARSHALLING

3.1 INTRODUCTION

Enormous volumes of goods are shipped yearly all over the world in standardized containers. These containers typically require multiple modes of transportation to reach their destination. At container terminals, containers are transshipped between ships, trucks, and trains. This transshipment generally does not occur immediately upon delivery of a container. Therefore, containers are temporarily stored in an area called the container yard. The container yard consists of a set of blocks, which in turn consist of a set of bays. Each bay contains a number of rows, called stacks, with a certain height.

One main indicator of the efficiency of a container terminal is the berthing time of a vessel, which consists primarily of the time needed to load and unload containers. During the unloading phase, information on pick-up times and destination of the containers is often inaccurate or even unknown. This makes it difficult to obtain an unloading sequence that permits an efficient loading sequence. Hence, during the loading phase it can occur that the container that needs to be retrieved next is not on top of its stack. In this case, the containers on top of the desired container need to be rehandled, i. e., relocated within the container yard, before the desired container can be retrieved. These rehandle operations greatly increase the time needed to retrieve the container from the yard.

One way to resolve this issue is to reshuffle the containers prior to the arrival of the vessel. This operation is called *remarshalling*. The goal of remarshalling is to find a minimum length sequence of rehandles that reorganizes the stacks such that no container that is needed early is below a container that is needed late. This results in no rehandles, also called moves, during the loading phase,

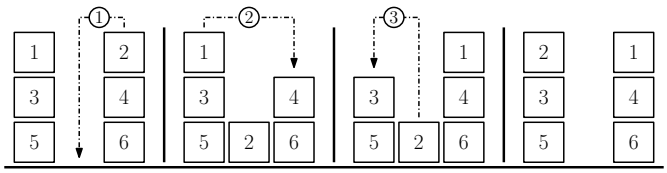

Figure 4: Small example of three moves swapping the position of containers 1 and 2.

thereby reducing the berthing time. The only valid move is to pick up the top container of one stack and put it on top of another (non-full) stack, see Figure 4 for an illustration.

Two types of remarshalling operations can be identified, namely *intra-block remarshalling* (hereafter called remarshalling) and *intra-bay premarshalling* (hereafter called premarshalling). The containers are reshuffled between bays for the former, and within a bay for the latter. The premarshalling variant is primarily applicable to container yards that use rail mounted gantry cranes. It is usually not allowed to move a container to another bay, as this operation is extremely time consuming [55].

Another main difference between remarshalling and premarshalling lies in the number of cranes that is used. For premarshalling typically only a single crane is used, while for remarshalling several cranes are often used simultaneously [14].

In this chapter we focus on the premarshalling problem, and we follow the same assumptions as Bortfeldt and Forster [11]: (a) a single crane is used for moving containers, and (b) the time needed to move a container from one stack to another does not depend on the distance between the two stacks. This last assumption follows from the fact that the time needed to position the crane over a stack is negligible compared to the time needed to pick up or drop off a container. As a consequence, we are only interested in the number of moves.

We assume that for each container a priority level is given, and the goal is to transform the initial lay-out into a desired target lay-out in the minimum number of moves. The main problem studied is called Priority Stacking: all lay-outs in which no container with a lower priority is placed on top of a container with a higher priority are accepted. In a variant, called Configuration Stacking, the

target lay-out is restricted to a single pre-specified lay-out. The main motivation to also look at CONFIGURATION STACKING is that giving a concrete target lay-out might give guidance to the algorithm and yield a faster computation time.

3.1.1 Related Literature

The operations at container terminals are well studied in the literature. Steenken, Voß, and Stahlbock [69] and Stahlbock and Voß [68] describe the most important processes and operations at container terminals and give an overview of methods to optimize these operations. Vis and De Koster [81] give a classification of the different decision problems that occur at container terminals, and give an overview of relevant literature. Lehnfeld and Knust [57] develop a classification scheme for loading, unloading and premarshalling problems that appear in several applications. This scheme is applied to the existing literature.

While there is a vast amount of work on the logistics of container terminals, the number of publications on the premarshalling problem is limited. We are only aware of one paper that thoroughly investigates an exact algorithm. Lee and Hsu [56] develop a mixed integer linear program based on a multicommodity network flow formulation that solves both PRIORITY STACKING and CONFIGURATION STACKING to optimality. However, this formulation can only be reasonably applied to very small instances, and the running time heavily depends on the choice for the number of allowed moves. For larger instances the authors provide a heuristic that iteratively applies the exact approach on small parts of the instance. Integer multicommodity network flow is a generalization of edge-disjoint paths which cannot be approximated better than $\Omega(\sqrt{n})$, which immediately implies that the integrality gap of this formulation is at least that [43].

The remaining literature on the premarshalling problem is on the design of fast heuristics for PRIORITY STACKING. Lee and Chao [55] describe a heuristic that minimizes the weighted sum of the mis-overlay index and the number of rehandles during the premarshalling phase. The mis-overlay index can be seen as a measure for the number of rehandles during the loading phase. Caserta and Voß [16] develop a heuristic based on the corridor method, where the basic idea is to use an exact method for limited portions of the entire solution space. Bortfeldt and Forster [11] describe a refined heuristic tree search procedure that looks

at move sequences rather than individual moves. This heuristic is reported to be faster than the heuristic by Caserta and Voß. Huang and Lin [46] describe two heuristics that iteratively improve the lay-out of the yard. The second heuristic is applied to a special case of PRIORITY STACKING, where all containers in a stack should be of the same priority level. Expósito-Izquierdo, Melián-Batista, and Morena-Vega [28] describe a heuristic that considers the containers from lowest to highest priority. The considered container is moved to a position where it is not above a container with higher priority. The authors also provide an instance generator and describe an A^* search algorithm that provides the optimal solution for smaller instances.

Caserta, Schwarze, and Voß [14] give an overview of recent developments on three so-called post-stacking problems. Besides the remarshalling and premarshalling problem, the authors also consider the (intra-bay) blocks relocation problem. In addition to the premarshalling problem, containers need to be removed from the bay in a certain order that minimizes the number of rehandles. It was proven that this problem is NP-hard, but with arbitrarily high stacks [15]. This proof can be easily adapted to show that PRIORITY STACKING with arbitrarily high stacks is also NP-hard. To the best of our knowledge there are no complexity results for the natural case where the stack height is bounded by a constant. Typical stack heights are between 2 and 8 containers, while currently used equipment can handle a stack height of at most 10 containers [68], [81].

3.1.2 Our Contributions

We develop a fast exact algorithm for the premarshalling problem that combines column generation and branch-and-bound. This algorithm is extensively evaluated. To the best of our knowledge, we are the first to extensively experiment with an exact algorithm. Expósito-Izquierdo, Melián-Batista, and Morena-Vega [28] describe an A^* search algorithm, but its results are only used as a benchmark for their heuristic. Lee and Hsu [56] also design an exact algorithm, but only evaluate it on two instances. Our algorithm is evaluated on 960 instances, with roughly 70% of the instances solved within one second. We also observe that our method exhibits a low integrality gap. Finally, we consider the complexity of PRIORITY STACKING and CONFIGURATION STACKING. Current NP-hardness proofs require a stack height that depends on the number of containers.

We strengthen this results by showing that both problems are already NP-hard for a fixed height of at least six containers, which is more in line with the real-life situation.

3.1.3 Organization

In Section 3.2 we introduce notation and formally describe the premarshalling problem. In Section 3.3 we consider the complexity of both premarshalling variants. In Section 3.4 we describe an ILP formulation and an oracle for finding variables in a column generation approach. This is then used in Section 3.5 to design a branch-and-price algorithm, whose experimental performance is analyzed in Section 3.6. Finally, conclusions are drawn in Section 3.7.

3.2 PRELIMINARIES

Let $[n]$ denote the set of integers from 1 up to n, i. e., $[n] := \{1, \ldots, n\}$. The premarshalling problem is defined as follows. Given are m stacks of maximum height h, and n containers labeled with a priority ℓ from $[k]$ $(2 \leqslant k \leqslant n)$. In line with the definitions used in the literature, a lower priority number indicates a higher priority level, i. e., containers with priority 1 are needed first, and containers with priority k last. The *lay-out* of a stack i with $j \leqslant h$ containers is denoted as an ordered set of priorities $X_i := (x_1, \ldots, x_j)$. The first element is the priority of the bottom container and the last element is the priority of the top container. Notice that containers with the same priority are indistinguishable from each other, therefore when the context is clear, we will abbreviate "container with priority ℓ" to "container ℓ".

The goal is to transform the initial lay-out into a target lay-out by performing the minimum number of moves, while adhering to the maximum stack height. A move is defined as picking up the top-most container of one stack and placing it on top of another stack. For Priority Stacking the set of target lay-outs consists of all lay-outs such that all stacks are sorted in non-increasing order when viewed from the bottom, i. e., for a stack i with $X_i := (x_1, \ldots, x_j)$, we have that $x_{p+1} \leqslant x_p$ for $p = 1, \ldots, j-1$. For Configuration Stacking there is only one target lay-out, which is specified beforehand.

3.3 COMPLEXITY

In this section we review the complexity status of PRIORITY STACKING and CONFIGURATION STACKING. Caserta, Schwarze, and Voß [15] show that the blocks relocation problem is NP-hard for *arbitrary* stack height. Their proof can also be used to show that PRIORITY STACKING is NP-hard for arbitrary stack height. We improve this result by showing that both PRIORITY STACKING and CONFIGURATION STACKING are NP-hard for a fixed stack height of at least six.

Both reductions are from the Mutual Exclusion Scheduling problem on permutation graphs, see Jansen [47]. Given a permutation $\pi : [n] \to [n]$, the permutation graph $G = ([n], E)$ has a vertex for each element in the set $[n]$, and there is an edge $(u, v) \in E$ for all $u < v$ if and only if $\pi(u) > \pi(v)$. Permutation graphs are exactly the class of graphs that are both *comparability graphs* and *co-comparability graphs*. Permutation graphs can also be viewed as an *intersection graph* of a set of line segments that connect two parallel lines.

For MUTUAL EXCLUSION SCHEDULING we are given permutation graph $G = ([n], E)$, and integers $k \geqslant 1$ and $m \geqslant 2$. The goal is to partition the vertex set into disjoint sets $P_1, \ldots, P_k$ with $|P_i| \leqslant m$ and P_i an independent set in G, for all $i \in [k]$. Notice that P_i is an independent set if and only if for all $u, v \in P_i$ such that $u < v$ we have that $\pi(u) < \pi(v)$. Jansen proved that even for fixed $m \geqslant 6$ the problem is NP-hard for permutation graphs.

Theorem 3.1 (**Theorem 1.1.** in [47]). *For each fixed* $m \geqslant 6$, MUTUAL EXCLUSION SCHEDULING *is NP-hard for permutation graphs.*

In the NP-hardness proofs we will use three types of containers, called *node*, *dummy*, and *fill* containers. For each vertex $i \in [n]$ of the permutation graph, there will be a node container i, which also has priority level i. All other containers will be either dummy or fill containers, where dummy containers have to be moved at least once, i. e., they are placed above a container with a higher priority, while fill containers do not necessarily have to be moved. In figures, the containers are depicted by square boxes. Node and dummy containers can be identified by their black or rounded corners, respectively. For a graphical representation of the types of containers, see Figure 5.

Figure 5: Types of containers

3.3.1 Priority Stacking is NP-hard for Fixed Height

We show that PRIORITY STACKING is NP-hard when the maximum stack height is fixed. Based on an instance for MUTUAL EXCLUSION SCHEDULING, we construct the instance for PRIORITY STACKING as follows. The maximum height h is equal to m. For each $j \in [k]$ there is a stack j which is initially empty. For each $i \in [n]$ there is a stack $s_{1,i}$. Stack $s_{1,1}$ initially consists of fill container $-n$ and node container $\pi(n)$. For the other stacks $s_{1,i}$ the initial lay-out consists of fill, node, and dummy container $-n+i-1$, $\pi(n-i+1)$, and $-n+i-2$, respectively. Finally, there are $z = km - n$ stacks, called $s_{2,i}$ for $i = 1, \ldots, z$, with as initial lay-out dummy container 0 on top of fill container $-\infty$. See Figure 6 for an illustration.

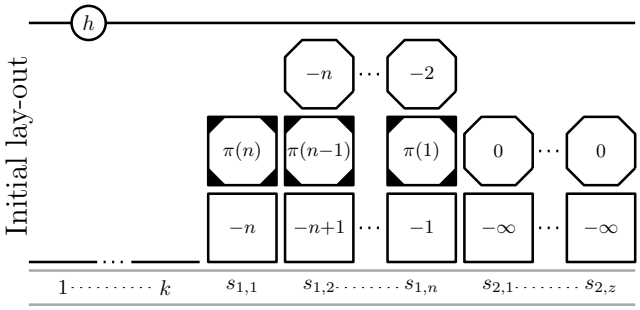

Figure 6: Illustration of PRIORITY STACKING instance

Theorem 3.2. *For every fixed $h \geqslant 6$,* PRIORITY STACKING *is NP-hard.*

Proof. Consider Figure 6, and observe that every node container is placed on top of a fill container with a higher priority. Furthermore, all dummy containers are either placed on a (wrongly placed) node container, or on a fill container with a higher priority. Hence, all node and dummy containers need to be moved at least once, so that the lower bound on the number of moves is equal to $z + 2n - 1$.

We show that a solution for MUTUAL EXCLUSION SCHEDULING exists if and only if a solution of $z + 2n - 1$ moves exists for PRIORITY STACKING.

Consider a solution for PRIORITY STACKING consisting of $z + 2n - 1$ moves. Hence, all node and dummy containers are moved exactly once, while no fill container is moved at all. We show that this solution exhibits the following two properties.

Property 3.3. *All node containers are placed on stacks* $1, \dots, k$ *after they are moved. Each of these stacks contains at most* m *node containers.*

Proof. Observe that the node containers can only be moved to stacks $1, \dots, k$, as all other stacks contain a fill container with a higher priority. Since the maximum stack height is m, by definition at most m node containers can be moved to any of these stacks. □

Property 3.4. *The node containers are moved in the order* $\pi(n), \dots, \pi(1)$.

Proof. Consider the z dummy containers with priority 0. As for the n node containers, these dummy containers can only be moved to stacks $1, \dots, k$. Observe that in total we have km node containers and dummy containers with priority 0, which coincides with the total number of spots available for stacks $1, \dots, k$. This implies that the dummy containers with priority $-2, \dots, -n$ cannot be moved to any of these stacks. These containers can also not be moved to stacks $s_{2,1}, \dots, s_{2,z}$, since they contain a fill container with priority $-\infty$. Hence, for $i = 1, \dots, n-1$, dummy container $-i-1$, which is initially placed on stack $s_{1,n-i+1}$, can only be moved to stacks $s_{1,n-i}$ and $s_{1,n-i+2}, \dots, s_{1,n}$, i. e., it can be moved one stack to the "left" or any $s_{1,j}$ stack to the "right". It cannot be moved more than one stack to the left, as that stack contains a fill container with a higher priority.

Assume to the contrary that Property 3.4 does *not* hold, i. e., the node containers are not moved in order $\pi(n), \dots, \pi(1)$. Let $\pi(i)$ denote the node container with highest index i that is moved before $\pi(i+1)$. From this we know that node containers $\pi(1), \dots, \pi(i-1)$ and $\pi(i+1)$ are moved after $\pi(i)$. Furthermore, $\pi(i)$ is moved after dummy container $-i-1$, since this dummy container is placed on top of $\pi(i)$. But then, we have that dummy container $-i-1$ can only be moved to a stack that still contains a node container. That would block this node container

from moving, preventing a target lay-out from being reached. This contradicts our assumption, so that Property 3.4 indeed holds. □

Applying these two properties, we arrive at the following lemma.

Lemma 3.5. *After all node containers are moved, the lay-out of stacks* $1, \ldots, k$ *constitutes a solution for* MUTUAL EXCLUSION SCHEDULING.

Proof. For $j \in [k]$, let P_j contain all nodes i such that node container $\pi(i)$ is placed on stack j. By Property 3.3 we know that $P_1, \ldots, P_k$ form a partition, and that $|P_j| \leqslant m$ for all j. Hence, we only need to show that each P_j is an independent set in the permutation graph.

Clearly, P_j is an independent set if it contains at most one element. Take an arbitrary j such that $|P_j| \geqslant 2$, and let $\pi(a)$ and $\pi(b)$, with $a < b$, be two arbitrary node containers on stack j. Hence, we have that $\{a, b\} \subseteq P_j$. As $a < b$, we have by Property 3.4 that container $\pi(b)$ was moved to stack j before container $\pi(a)$, so that $\pi(a)$ is placed above $\pi(b)$. As we obtain a target lay-out for PRIORITY STACKING, we have that the stack is sorted in non-increasing order when viewed from the bottom. This implies that $\pi(a) < \pi(b)$, so that there is no edge between nodes a and b in the permutation graph. As the two nodes were chosen arbitrarily, it holds for all pairs of nodes, so that P_j is an independent set. As j was also chosen arbitrarily, it holds for all P_j, proving the lemma. □

By Lemma 3.5 we have a solution for MUTUAL EXCLUSION SCHEDULING if there is a solution for PRIORITY STACKING consisting of $z + 2n - 1$ moves.

Assume that a solution for MUTUAL EXCLUSION SCHEDULING exists. Consider the following sequence of moves for PRIORITY STACKING. Start with moving node container $\pi(n)$ to stack j, where j is such that $n \in P_j$. Then, for $i = 2, \ldots, n$, repeat the following two steps: (1) move dummy container $-n+i-2$ from stack $s_{1,i}$ to $s_{1,i-1}$, i.e., move it one stack to the "left", and (2) move node container $\pi(n-i+1)$ to stack j such that $n-i+1 \in P_j$. Finally, for $i \in [z]$, move the dummy container on stack $s_{2,i}$ to the "leftmost" stack among $1, \ldots, k$ that is not yet full. Observe that this sequence satisfies Property 3.4.

Note that the order in which the containers are moved is from left to right and then from top to bottom, and all containers are moved to the left. Hence, no moved container will ever block a future move. Also no stack will ever contain more than m containers, so that the sequence of moves is valid.

Consider the final lay-out. Stacks $s_{1,1}, \ldots, s_{2,z}$ contain either one container, or two containers with the same priority, which are both target lay-outs. For stacks $1, \ldots, k$, we have that first the node containers are placed, and then the dummy containers with priority 0. As these dummy containers have a higher priority than the node containers, we have that an entire stack is sorted correctly if the node containers in that stack are sorted correctly.

Lemma 3.6. *After all node containers are moved, the node containers in stacks* $1, \ldots, k$ *are sorted in non-increasing order when viewed from the bottom.*

Proof. Take an arbitrary stack j. If this stack contains at most one node container, then the lemma trivially holds. So, furthermore assume that j contains at least two node containers. Let $\pi(a)$ and $\pi(b)$, with $a < b$, be two arbitrary node containers in this stack. By construction, we have that $\{a, b\} \subseteq P_j$, which implies that $\pi(a) < \pi(b)$. Otherwise there would be an edge between a and b, contradicting that P_j forms an independent set. By Property 3.4 we also know that $\pi(b)$ was moved to stack j before $\pi(a)$, so that $\pi(a)$ is placed above $\pi(b)$. This immediately implies that $\pi(a)$ and $\pi(b)$ are sorted in the correct order, as $\pi(a) < \pi(b)$. As the two node containers were chosen arbitrarily, it holds for all pairs of node containers, so that the lemma holds for stack j. As j was also chosen arbitrarily, the lemma holds for all stacks $1, \ldots, k$. □

Hence, all stacks exhibit a target lay-out, so that we also have a solution for Priority Stacking consisting of $z + 2n - 1$ moves if we have a solution Mutual Exclusion Scheduling.

Combining all results shows that for every fixed $h \geqslant 6$ Priority Stacking is NP-hard. □

3.3.2 Configuration Stacking is NP-hard for Fixed Height

In the previous section we showed that PRIORITY STACKING is NP-hard for a fixed stack height. In this section we show that CONFIGURATION STACKING is also NP-hard if the maximum height is a constant with value at least six.

Based on an instance for MUTUAL EXCLUSION SCHEDULING, we construct the instance for CONFIGURATION STACKING as follows. The maximum height h is equal to m and there are $4n+k+2$ stacks, which can be split into five groups. For each vertex $i \in [n]$ in the permutation graph there is a node container i and dummy containers a_i to h_i, except for container h_n. Furthermore there are dummy containers x and y, and $z = (n+1)(h-3)$ fill container. All fill containers have priority ∞, while all other containers have a unique priority.

First, consider the initial lay-out. Stacks $1, \ldots, k$ are empty. Stacks $s_{1,1}, \ldots, s_{1,n}$ consist of five containers, where the lay-out of stack $s_{1,i}$ is $(g_{\pi(n-i+1)}, d_i, b_i, \pi(n-i+1), a_i)$. Stack $s_{2,1}$ contains e_1, and for $i = 2, \ldots, n$, stack $s_{2,i}$ consists of containers h_{i-1} and e_i. Stacks $s_{3,0}, \ldots, s_{3,n}$ contain $h-3$ fill containers each. For $i = 1, \ldots, n-1$, stack $s_{3,i}$ additionally contains container c_i, while stack $s_{3,n}$ additionally consists of containers x and c_n. Stack $s_{4,0}$ is empty, and stack $s_{4,i}$ consists of f_i for $i = 1, \ldots, n-1$. Stack $s_{4,n}$ contains containers y and f_n.

Second, consider the target lay-out. Stacks $1, \ldots, k$ and $s_{1,1}, \ldots, s_{1,n}$ are empty. The lay-out for stacks $s_{2,1}$ and $s_{2,n}$ is $(y, g_1, 1, h_1)$ and (g_n, n), respectively. Stack $s_{2,i}$ consists of containers g_i, i, and h_i, for $i = 2, \ldots, n-1$. Stacks $s_{3,0}, \ldots, s_{3,n}$ contain $h-3$ fill containers each. For $i = 0, \ldots, n-1$, stack $s_{3,i}$ additionally consists of dummy containers a_{i+1}, b_{i+1}, and c_{i+1}. Stack $s_{4,0}$ consists of 4 containers, namely x, d_1, e_1, and f_1. For $i = 1, \ldots, n-1$, the lay-out of stack $s_{4,i}$ is $(d_{i+1}, e_{i+1}, f_{i+1})$. Finally, stack $s_{4,n}$ is empty.

See Figure 7 for an illustration of the initial and target lay-out. For the node and dummy containers the name is given, while for the fill containers the priority (∞) is given.

Theorem 3.7. *For every fixed $h \geqslant 6$,* CONFIGURATION STACKING *is NP-hard.*

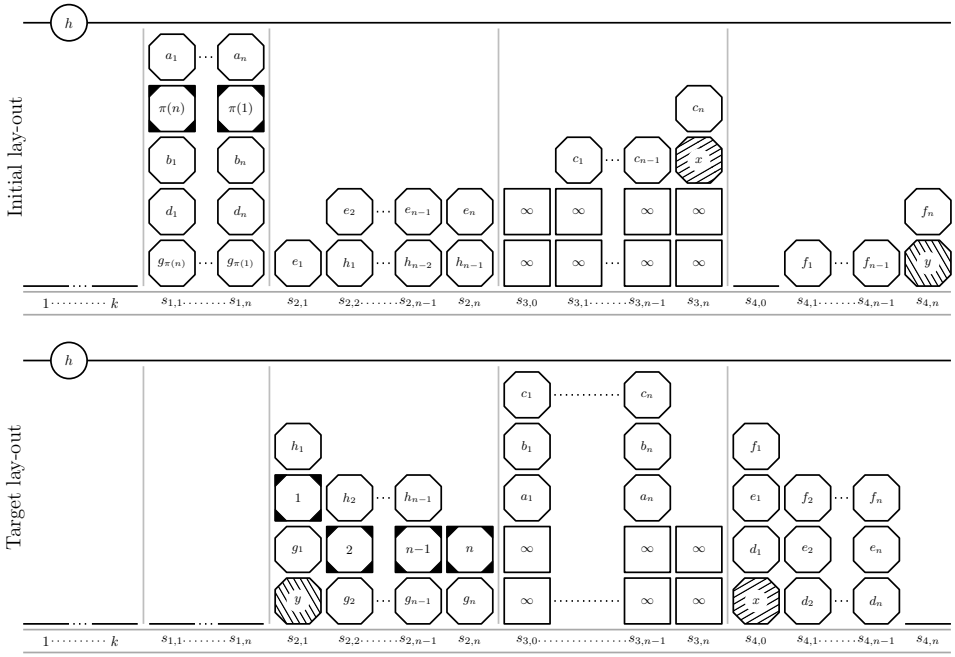

Figure 7: Illustration of CONFIGURATION STACKING instance

Proof. Take an arbitrary node container i, and observe that it is placed on top of dummy container g_i in both the initial and target lay-out. This implies that i has to be moved at least twice, once before g_i can be moved, and once after g_i is moved. As i was chosen arbitrarily, this holds for all node containers. Furthermore, all dummy containers need to be moved at least once, so that the lower bound on the number of moves for this instance is $10n + 1$.

Assume there is a solution for CONFIGURATION STACKING consisting of $10n + 1$ moves. This implies that the node containers are moved twice and the dummy containers once. We claim that there is only a single order of moves that transforms the initial lay-out into the target lay-out in $10n + 1$ moves. To see why this is true, let t_α denote the time point at which dummy container α is moved, and let $t_i^{(1)}$ and $t_i^{(2)}$ denote the first and second time point at which node container i is moved. Consider the following observations.

Observation 3.8. $t_{a_i} < t^{(1)}_{\pi(n-i+1)} < t_{b_i} < t_{c_i}$, $\forall i \in [n]$, *and* $t_{c_i} < t_{a_{i+1}}$, $\forall i \in [n-1]$.

Proof. The first two inequalities follow from the fact that a_i is placed on top of $\pi(n-i+1)$, which in turn is placed on top of b_i, in the initial lay-out of stack $s_{1,i}$. The third inequality follows from the target lay-out of stack $s_{3,i-1}$, where c_i is placed on top of b_i. The last inequality follows from the fact that a_{i+1} has to be moved to the initial position of c_i. This is only possible if c_i is moved before a_{i+1}. □

Observation 3.9. $t_{c_n} < t_x < t_{d_1}$.

Proof. The fist inequality follows from the initial lay-out of stack $s_{3,n}$, where c_n is placed on top of x. The second inequality follows from the target lay-out of stack $s_{4,0}$, where d_1 is placed on top of x. □

Observation 3.10. $t_{d_i} < t_{e_i} < t_{f_i}$, $\forall i \in [n]$, *and* $t_{f_i} < t_{d_{i+1}}$, $\forall i \in [n-1]$.

Proof. The first two inequalities follow from the target lay-out of stack $s_{4,i-1}$. The last inequality follows from the fact that d_{i+1} is moved to the initial position of f_i, which thus has to be moved first. □

Observation 3.11. $t_{f_n} < t_y < t_{g_1}$.

Proof. These inequalities follow from the initial lay-out of stack $s_{4,n}$ and the target lay-out of stack $s_{2,1}$, respectively. □

Observation 3.12. $t_{g_i} < t^{(2)}_i$, $\forall i \in [n]$, *and* $t^{(2)}_i < t_{h_i} < t_{g_{i+1}}$, $\forall i \in [n-1]$.

Proof. The first two inequalities follow from the target lay-out of stack $s_{2,i}$. The last inequality follows from the fact that g_{i+1} takes over the position of h_i. This is only possible if h_i is moved before g_{i+1}. □

Combining these observations, we can split the unique sequence of moves into three phases. In the first phase, the following set of moves is repeated for $i \in [n]$: move a_i to its final location in stack $s_{3,i-1}$, move $\pi(n-i+1)$ to an intermediate

stack, move b_i on top of a_i, and move c_i on top of b_i. This sequence follows immediate from Observation 3.8. The first phase is concluded by moving container x from stack $s_{3,n}$ to stack $s_{4,0}$, which follows from Observation 3.9.

In the second phase, we repeat the following set of moves for $i \in [n]$: move d_i to stack $s_{4,i-1}$, move e_i on top of d_i, and move f_i on top of e_i. This sequence follows from Observation 3.10. The second phase is concluded by moving container y from stack $s_{4,n}$ to stack $s_{2,1}$. This move follows from Observation 3.11.

In the third phase, repeat these set of moves for $i \in [n-1]$: move g_i to its final position, move i on top of g_i, and move h_i on top of i. Finally, move g_n to stack $s_{2,n}$, and place n on top of it. This sequence follows from Observation 3.12.

For this sequence, we will show that the following two properties, that resemble Properties 3.3 and 3.4, hold.

Property 3.13. *All node containers are positioned on stacks* $1, \ldots, k$ *after they are moved for the first time. Each of these stacks contains at most* m *node containers.*

Property 3.14. *The node containers are first moved in the order* $\pi(n), \ldots, \pi(1)$*. For their second move, the order is* $1, \ldots, n$*, where the second move of* 1 *occurs after the first move of* $\pi(1)$*.*

Proof. Observe that Property 3.14 immediately follows from the unique order of moves. To see that Property 3.13 also holds is not so trivial. Since the stack height is m, clearly at most m node containers are placed on any of the stacks $1, \ldots, k$. Hence, we need to show that all node containers are placed on stacks $1, \ldots, k$ after they are moved once. Thereto, take an arbitrary node container $\pi(i)$. Observe that stacks that are involved in a move that occurs between the first and second move of $\pi(i)$ cannot be used as intermediate stack, as it would block a move. This immediately excludes stacks $s_{1,1}, \ldots, s_{1,n}$, $s_{2,1}, \ldots, s_{2,n}$ and $s_{4,0}, \ldots, s_{4,n}$ for use as an intermediate stack. Consider the lay-out at the moment that $\pi(i)$ is moved for the first time. In previous moves $c_1, \ldots, c_{n-i}$ are moved to their final position, so that currently stacks $s_{3,0}, \ldots, s_{3,n-i-1}$ are full. In future moves, but before $\pi(i)$ is moved for the second time, $c_{n-i+1}, \ldots, c_n$ need to be moved to their final position. These moves involve stacks $s_{3,n-i}, \ldots, s_{3,n}$, which implies that stacks $s_{3,0}, \ldots, s_{3,n}$ can also not be used as intermediate stacks. As $\pi(i)$ was

chosen arbitrarily, this holds for all node containers. This leaves stacks $1, \ldots, k$ as the only candidates for intermediate stacks, so that Property 3.13 also holds. □

Applying Properties 3.13 and 3.14, we state the following lemma, that resembles Lemma 3.5.

Lemma 3.15. *After all node containers are moved once, the lay-out of stacks* $1, \ldots, k$ *constitutes a solution for* MUTUAL EXCLUSION SCHEDULING.

Proof. The proof is identical to the one for Lemma 3.5. The stacks still need to be sorted in non-increasing order when viewed from the bottom. This time *not* to obtain a target lay-out, but to make sure that the order for the second move of the node containers, namely $1, \ldots, n$, is possible. □

Hence, by Lemma 3.15 we have a solution for MUTUAL EXCLUSION SCHEDULING if we have a solution for CONFIGURATION STACKING consisting of $10n + 1$ moves.

Assume we have a solution for MUTUAL EXCLUSION SCHEDULING. As solution for CONFIGURATION STACKING we take the same order of moves as mentioned above. For the dummy containers, which are only moved once, the exact move is thus known, as we know the origin and destination stack from the initial and target lay-out, respectively. For the node containers, which are moved twice, we have to specify an intermediate stack. As intermediate stack for $\pi(i)$ we take stack j such that $i \in P_j$. If all moves are valid, we obtain the target lay-out by definition.

As all node containers are moved to stacks $1, \ldots, k$, they do not "interfere" with the moves of the dummy containers, i. e., node containers are not moved on top of a dummy container. Hence, all moves involving dummy containers are valid, so that we only need to show that the moves involving the node containers are valid. Clearly, the first move of each node container is valid, since at most m node containers are placed on each stack. The second move of the node containers, which is in the order $1, \ldots, n$, is possible if stacks $1, \ldots, k$ are sorted in non-increasing order. Therefore, we state the following lemma, which resembles Lemma 3.6.

Lemma 3.16. *After all node containers are moved once, the node containers in stacks* $1, \ldots, k$ *are sorted in non-increasing order when viewed from the bottom.*

Proof. The proof is identical to the one for Lemma 3.6. □

Hence, all moves are valid and we obtain the target lay-out, so that we have a solution for CONFIGURATION STACKING containing $10n + 1$ moves if we have a solution for MUTUAL EXCLUSION SCHEDULING.

Concluding, we have that for every fixed $h \geqslant 6$ CONFIGURATION STACKING is NP-hard. □

3.4 ILP FORMULATION AND SEPARATION ORACLE

In this section we first describe the linear program model that we use for both PRIORITY STACKING and CONFIGURATION STACKING. After that we explain our separation oracle for finding columns with negative reduced costs.

3.4.1 ILP Formulation

Before we describe the ILP formulation, let us first introduce some notation. Let T denote the number of time points. As only a single move is allowed per time point, T can also be viewed as the maximum possible number of moves. Let the tuple (ℓ, t) denote a move of container $\ell \in [k]$ at time $t \in [T]$. Consider stack $s \in [m]$. Let L_s^{add} and L_s^{rem} contain moves (ℓ, t) such that container ℓ is respectively added to, or removed from, stack s at time t, and let $L_s := (L_s^{\mathsf{add}}, L_s^{\mathsf{rem}})$. The set L_s is *feasible* if (a) at every time t there is at most one move, i. e., either a container is added, a container is removed, or no move is performed; (b) for all tuples $(\ell, t) \in L_s^{\mathsf{add}}$ we have that at time t stack s contains at least one free spot; (c) for all tuples $(\ell, t) \in L_s^{\mathsf{rem}}$ we have that at time t container ℓ is the top container of stack s; (d) the lay-out obtained by executing the moves in order of their time points constitutes a target lay-out. Hence, a feasible L_s can be viewed as a sequence of moves that transforms stack s into a target lay-out, where L_s^{add} and L_s^{rem} consist of the moves where a container is added or removed, respectively. Furthermore, let $\#\mathsf{moves}(L_s)$ denote the number of moves in L_s. As each move consists of removing a container from one stack and adding it to another stack, we define the number of moves as the number of added containers, i. e., $\#\mathsf{moves}(L_s) := |L_s^{\mathsf{add}}|$. Finally, let $\mathcal{L}_s$ denote the set of all sequences L_s that trans-

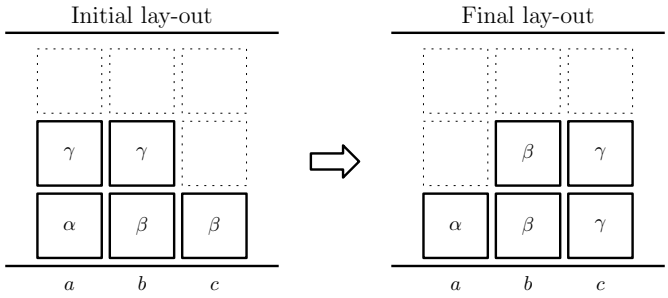

Figure 8: Example illustrating a feasible set of sequences. For ease of reference, the stacks are called a, b, and c, and the priorities α, β, and γ. The final lay-out is obtained by moving container β from stack c to stack a, γ from b to c, β from a to b, and γ from a to c. Let L_a, L_b, and L_c denote the sequence for stack a, b, and c, respectively. For the mentioned set of moves we get $L_a^{\mathsf{add}} = \{(\beta, 1)\}$, $L_a^{\mathsf{rem}} = \{(\beta, 3), (\gamma, 4)\}$, $L_b^{\mathsf{add}} = \{(\beta, 3)\}$, $L_b^{\mathsf{rem}} = \{(\gamma, 2)\}$, $L_c^{\mathsf{add}} = \{(\gamma, 2), (\gamma, 4)\}$, and $L_c^{\mathsf{rem}} = \{(\beta, 1)\}$. For the number of moves we get $\#\mathsf{moves}(L_a) = \#\mathsf{moves}(L_b) = 1$ and $\#\mathsf{moves}(L_c) = 2$.

form stack s into a target lay-out, i. e., $\mathcal{L}_s := \{L_s : L_s \text{ is feasible}\}$. For an example of a feasible set of sequences, see Figure 8.

We consider the following integer linear program ILP, where $x_{s,L} \in \{0, 1\}$ for $s \in [m]$ and $L \in \mathcal{L}_s$ has value 1 if and only if stack s is transformed according to sequence L. Let $\mathsf{add}(L, \ell, t)$ and $\mathsf{rem}(L, \ell, t)$ be equal to 1 if and only if $(\ell, t) \in L^{\mathsf{add}}$ and $(\ell, t) \in L^{\mathsf{rem}}$, respectively.

$$\min \quad \sum_{s \in [m]} \sum_{L \in \mathcal{L}_s} \#\mathsf{moves}(L)\, x_{s,L} \tag{ILP}$$

$$\text{s.t.} \quad \sum_{s \in [m]} \sum_{L \in \mathcal{L}_s} \Big(\mathsf{add}(L, \ell, t) - \mathsf{rem}(L, \ell, t)\Big) x_{s,L} \geqslant 0, \qquad \forall\, \ell \in [k], t \in [T], \tag{C1}$$

$$\sum_{s \in [m]} \sum_{L \in \mathcal{L}_s} \sum_{\ell \in [k]} \mathsf{add}(L, \ell, t) x_{s,L} \leqslant 1, \qquad \forall\, t \in [T], \tag{C2}$$

$$\sum_{L \in \mathcal{L}_s} x_{s,L} = 1, \qquad \forall\, s \in [m], \tag{C3}$$

$$x_{s,L} \in \{0, 1\}, \qquad \forall\, s \in [m], L \in \mathcal{L}_s.$$

The variables already ensure that only sequences of moves are chosen that *for every stack individually* are feasible. The remaining constraints ensure that the locally feasible solutions together form a valid global solution. Constraint (C1) ensures that at time t the number of containers of priority ℓ that are added is at least the number of containers of priority ℓ that are removed. Constraint (C2) enforces that at any time point t at most one container can be added. Note that this implies that at most one container is moved per time point. Constraint (C3) makes sure that exactly one sequence is selected for each stack. By relaxing the requirement that the variables are either 0 or 1 we obtain the relaxation LP Primal.

$$\begin{aligned}
\min \quad & \sum_{s\in[m]} \sum_{L\in\mathcal{L}_s} \#\mathsf{moves}(L)\, x_{s,L} && \text{(LP Primal)}\\
\text{s.t.} \quad & \text{Constraints (C1), (C2), and (C3),} \\
& x_{s,L} \geqslant 0, && \forall\, s\in[m], L\in\mathcal{L}_s.
\end{aligned}$$

To obtain the optimal solution for LP Primal we apply column generation. Thereto, let $\alpha_{\ell,t}$, β_t, and γ_s be the dual variables for constraints of type (C1), (C2) and (C3), respectively. The dual LP is as follows.

$$\begin{aligned}
\max \quad & \sum_{t\in[T]} -\beta_t + \sum_{s\in[m]} \gamma_s && \text{(LP Dual)}\\
\text{s.t.} \quad & \sum_{\ell\in[k]} \sum_{t\in[T]} \Big(\mathsf{add}(L,\ell,t) - \mathsf{rem}(L,\ell,t)\Big)\alpha_{\ell,t} - \\
& \qquad \sum_{\ell\in[k]} \sum_{t\in[T]} \mathsf{add}(L,\ell,t)\beta_t + \gamma_s \leqslant \#\mathsf{moves}(L), && \forall\, s\in[m], L\in\mathcal{L}_s, \quad \text{(D1)}\\
& \alpha_{\ell,t} \geqslant 0, && \forall\, \ell\in[k], t\in[T],\\
& \beta_t \geqslant 0, && \forall\, t\in[T],\\
& \gamma_s \text{ unrestricted}, && \forall\, s\in[m].
\end{aligned}$$

3.4.2 Finding Columns with Negative Reduced Costs

To apply column generation we need an algorithm that finds sequences with negative reduced costs. Given the dual variables $\alpha_{\ell,t}$, β_t, and γ_s, the goal is to find (if any) $L \in \mathcal{L}_s$ with negative reduced costs, i. e., for a specific stack s the goal is to find a feasible sequence L that maximizes

$$\sum_{\ell\in[k]}\sum_{t\in[T]}\Big(\mathsf{add}(L,\ell,t)-\mathsf{rem}(L,\ell,t)\Big)\alpha_{\ell,t}- \sum_{\ell\in[k]}\sum_{t\in[T]}\mathsf{add}(L,\ell,t)\beta_t+\gamma_s-\#\mathsf{moves}(L),$$

which can be rewritten as

$$\sum_{\ell\in[k]}\sum_{t\in[T]}\mathsf{add}(L,\ell,t)(\alpha_{\ell,t}-\beta_t-1)-\sum_{\ell\in[k]}\sum_{t\in[T]}\mathsf{rem}(L,\ell,t)\alpha_{\ell,t}+\gamma_s.$$

Observe that adding a container with priority ℓ at time t contributes $\alpha_{\ell,t}-\beta_t-1$ towards this expression, while removing a container with priority ℓ at time t contributes $-\alpha_{\ell,t}$. We introduce intervals $[t_1,t_2;\ell]$, which indicate that a container with priority ℓ is added to the stack at time t_1 and removed from the stack at time t_2. Each interval $[t_1,t_2;\ell]$ has a corresponding weight $w[t_1,t_2;\ell]$.

Consider an arbitrary stack s. Since γ_s is constant for stack s, we disregard it for now. We phrase the goal of finding a maximum weight sequence for this stack as finding a maximum weight set of intervals I such that (a) all intervals are non-overlapping, and (b) there is no time point t such that more than h intervals from I intersect with t. This second condition ensures that the maximum stack height h is not violated. Two intervals $[a,b]$ and $[c,d]$ *overlap* if $[a,b]\cap[c,d]\neq\emptyset$ and neither interval contains the other interval. A set of intervals is *non-overlapping* if the intervals pairwise do not overlap.

A set of intervals that adheres to these two conditions is called *feasible*. Note that the collection of all feasible sets of intervals exactly represents the set of all feasible sequences $\mathcal{L}_s$. To see why this is true, consider the pair of intervals $[s_1,s_2;\sigma]$ and $[t_1,t_2;\tau]$. Suppose that these two intervals do not overlap. Hence, we have that either they do not intersect, or one interval contains the other. Without loss of generality, assume that for the first case we have $s_1<s_2<t_1<t_2$ and for the second case $s_1<t_1<t_2<s_2$. For the first case this corresponds to adding

container σ, removing σ, adding τ, and removing τ, while for the second case this corresponds to adding container σ, adding τ, removing τ and removing σ. Clearly, both sequences of moves are feasible. Now, suppose that the two intervals overlap, and without loss of generality assume that $s_1 < t_1 < s_2 < t_2$. This would correspond to adding container σ, adding τ, removing σ, and removing τ. This sequence of moves is not feasible, as we try to remove σ while τ is still placed on top of σ.

Without condition (b) this problem would be equivalent to finding a maximum weight independent set in a circle graph, which can be solved in polynomial time by dynamic programming [10, 75]. A circle graph is an intersection graph of chords of a circle, and two vertices/chords are adjacent if and only if they intersect. Circle graphs can equivalently be defined as the overlap graph of a set of intervals. Note that the dynamic program to find the maximum weight independent set of intervals can be easily adapted to take also condition (b) into account.

3.4.2.1 *Intervals for* PRIORITY STACKING

To define all relevant intervals for PRIORITY STACKING, we consider time points $-h+1, \dots, T+kh$. Time points $-h+1, \dots, 0$ are used to represent the initial lay-out, and time points $T+1, \dots, T+kh$ to represent the final lay-out. Time points $1, \dots, T$ are the regular time points, where one container can be moved per time point. Hence, for intervals whose start point is in the range $-h+1, \dots, 0$, the corresponding container is not added to the stack but is rather part of the initial lay-out. Similarly, for intervals whose end point is in the range $T+1, \dots, T+kh$, the corresponding container is not removed from the stack but is rather part of the final lay-out. Using these time points, we define the following set of intervals I and corresponding weights w.

- *Containers with priority ℓ that are initially at height i, and which are not moved:* $[i{-}h, T{+}(\ell{-}1)h{+}j; \ell]$ for all $1 \leqslant j \leqslant h$ with $w[i{-}h, T{+}(\ell{-}1)h{+}j; \ell] = 0 + 2B$.
- *Containers with priority ℓ that are initially at height i, and that are later removed:* $[i{-}h, t; \ell]$ for all $1 \leqslant t \leqslant T$ with $w[i{-}h, t; \ell] = -\alpha_{\ell,t} + 2B$.

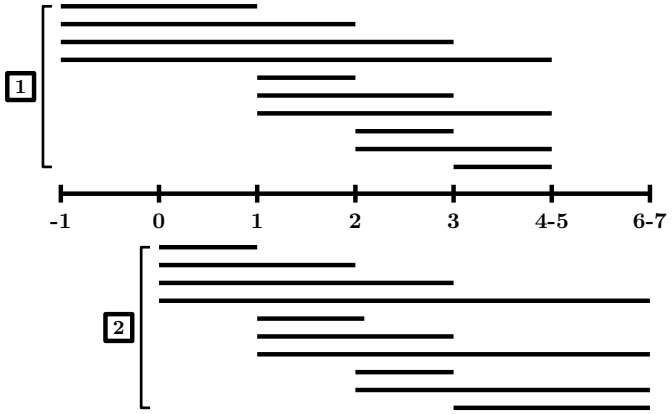

Figure 9: Example set of intervals for an instance for PRIORITY STACKING, consisting of containers with priority 1 or 2. The initial lay-out consists of a container with priority 1 at height 1 and a container with priority 2 at height 2. The corresponding intervals for $T = 3$ moves and maximum height $h = 2$ are depicted. Above the time line the intervals for containers with priority 1 are shown, while below the time line the intervals corresponding to containers with priority 2 are depicted. Time points 4 and 5 (6 and 7) are depicted as a single time point. Observe that for every interval ending at time point 4 (6), there is a corresponding interval ending at time point 5 (7) with identical weight. Also note that the intervals starting at time -1 or 0 are exactly the heavy intervals.

- *Containers with priority ℓ that are added to the stack and never removed:* $[s, T+(\ell-1)h+j; \ell]$ for all $1 \leqslant s \leqslant T$ and $1 \leqslant j \leqslant h$ with $w[s, T+(\ell-1)h+j; \ell] = \alpha_{\ell,s} - \beta_s - 1$.
- *Containers with priority ℓ that are added to the stack and later removed:* $[s, t; \ell]$ for all $1 \leqslant s < t \leqslant T$ with $w[s, t; \ell] = \alpha_{\ell,s} - \alpha_{\ell,t} - \beta_s - 1$.

Note that B is added to represent the initial lay-out of the stack. The value of B is equal to the sum of all $\alpha_{\ell,t}$ and β_t values, and intervals with an added weight of $2B$ are called *heavy* intervals. For an illustration of these intervals, see Figure 9.

Lemma 3.17. *A maximum weight feasible set of intervals represents a maximum weight sequence.*

Proof. Disregarding the added weight of $2B$ for the intervals pertaining to the initial solution, we have that any feasible set of intervals has a weight ranging between $-B$ and B. Hence, adding a heavy interval at the expense of other non-heavy intervals always increases the weight of the selected set of intervals. Since all heavy intervals start in the range $-h+1,\ldots,0$ and at most one selected interval may start per time point, at most h heavy intervals can be selected. Combining these observations, exactly h heavy intervals will be selected, one for each time point in the range $-h+1,\ldots,0$, so that for every container in the initial lay-out a corresponding interval is selected.

Without loss of generality assume that the initial lay-out is of height h and let $X := (x_1,\ldots,x_h)$ denote this initial lay-out. Consider containers x_i and x_{i+1} from the initial lay-out. All intervals pertaining to the initial positions of x_i and x_{i+1} start at time point $i-h$ and $i-h+1$, respectively. Hence, the start point for intervals for x_i is to the "left" of the start point for intervals for x_{i+1}. Assume that for both containers an interval corresponding to its initial position is chosen. Since these intervals are not allowed to overlap, we must have that the end point of the interval for x_i is to the "right" of the end point for x_{i+1}. This implies that x_{i+1} is removed first, and x_i is removed thereafter. This is indeed correct, as x_{i+1} is placed above x_i.

With a similar reasoning we can show that containers with a higher priority are placed above containers with a lower priority in the final lay-out. Consider two containers ℓ and $\ell+1$, for which intervals with end point $T+(\ell-1)h+i$ and $T+\ell h+j$ are selected, respectively. Observe that both i and j have a value in the range $1,\ldots,h$, so that $T+(\ell-1)h+i$ is to the "left" of $T+\ell h+j$. Again by the condition that intervals are not allowed to overlap, we must have that the interval selected for container $\ell+1$ starts earlier then the interval selected for container ℓ. This implies that $\ell+1$ is added first, so that ℓ is placed on top of $\ell+1$. This corresponds to a desired final lay-out. □

3.4.2.2 *Dynamic Program for* Priority Stacking

Having defined the set of intervals, we can now set up a dynamic program to find the maximum weight feasible set of intervals. Let $I[a,b]$ contain the intervals that are a subset of the range $[a,b]$, i. e., $I[a,b] := \{[c,d;\ell] \in I : [c,d] \subseteq [a,b]\}$. For

any interval $[a, b]$ and remaining height j we consider all "leftmost" intervals $[a, c]$, with $a < c \leqslant b$. A leftmost interval either covers no container, or it covers a container with some priority ℓ. If it covers no container, then no containers are added to or removed from this stack in this interval. If the interval covers container ℓ, other containers can be added to or removed from the stack in the interval $[a + 1, c - 1]$. Clearly, containers can always be added to or removed from the stack in the remaining rightmost interval.

Consider state $A[a, b, j]$, which represents the optimal solution restricted to interval $[a, b]$ and remaining height j. Both a and b are restricted to the extended set of time points, i. e., $a, b \in \{-h + 1, \ldots, T + kh\}$. The remaining height j is restricted to the set $\{0, \ldots, h\}$. By definition we have that $A[a, b, j] := 0$ if $b \leqslant a$ or $j = 0$ or $|I[a, b]| = 0$. For all other states we apply the following recurrence relation

$$A[a, b, j] := \max_{a<c\leqslant b} \begin{cases} A[c, b, j] \\ \max\limits_{i=[a,c;\ell]\in I[a,c]} \Big\{ w(i) + A[a+1, c-1, j-1] + A[c+1, b, j] \Big\}. \end{cases}$$

The optimal solution is represented by state $A[-h + 1, T + kh, h]$. There are in total $(T + (k + 1)h)^2(h + 1)$ states, and evaluating a single state in the dynamic program takes at most $O((T + (k + 1)h)(k + 1))$ time. Hence, the running time is polynomial.

3.4.2.3 *Intervals and Dynamic Program for* Configuration Stacking

The set of intervals for Configuration Stacking is similar to the set of intervals for Priority Stacking. The only difference is that we now want to obtain a specific target lay-out. This is achieved in the same manner as that the initial lay-out is enforced. Hence, in stead of using time points $T + 1, \ldots, T + kh$ to represent the final lay-out, it now suffices to use time points $T + 1, \ldots, T + h$. Without loss of generality assume that the target lay-out is also of height h, and let $Y := (y_1, \ldots, y_h)$ denote the target lay-out. Intervals corresponding to container y_i have $t + h - i + 1$ as end point. Using these time points, we define the following set of intervals I and corresponding weights w.

- *Containers with priority ℓ that are at height i in the initial and target lay-out, and which are not moved:*
  $[i{-}h, T{+}h{-}i{+}1; \ell]$ with $w[i{-}h, T{+}h{-}i{+}1; \ell] = 0 + 4B$.
- *Containers with priority ℓ that are initially at height i, and that are later removed:*
  $[i{-}h, t; \ell]$ for all $1 \leqslant t \leqslant T$ with $w[i{-}h, t; \ell] = -\alpha_{\ell,t} + 2B$.
- *Containers with priority ℓ that are added to the stack and never removed, and are at height i in the target lay-out:*
  $[s, T{+}h{-}i{+}1; \ell]$ for all $1 \leqslant s \leqslant T$ with $w[s, T{+}h{-}i{+}1; \ell] = \alpha_{\ell,s} - \beta_s - 1 + 2B$.
- *Containers with priority ℓ that are added to and later removed from the stack:*
  $[s, t; \ell]$ for all $1 \leqslant s < t \leqslant T$ with $w[s, t; \ell] = \alpha_{\ell,s} - \alpha_{\ell,t} - \beta_s - 1$.

The dynamic program for CONFIGURATION STACKING is identical to the one for PRIORITY STACKING. The only difference is that state $A[a, b, j]$ is now only defined for $a, b \in \{-h+1, \ldots, T+h\}$. As a result, we have that the optimal solution is now represented by state $A[-h+1, T+h, h]$. The running time remains polynomial. Only $(T+2h)^2(h+1)$ states need to be considered, and evaluating a single state takes at most $O((T+2h)(k+1))$ time.

3.5 BRANCH-AND-PRICE ALGORITHM

To solve the premarshalling problem, several branch-and-price trees with a different number of time points are considered. For the first tree the number of time points is equal to some lower bound. If in this tree a feasible solution is found, then the algorithm is terminated. If no feasible solution is found, then the number of time points (and thus also the number of allowed moves) is increased by one. This procedure is repeated until a feasible solution is found. See Algorithm 1 for an overview.

In the following sections we first describe our branching rule. One important feature of the branching rule is that it should not increase the difficulty of the pricing problem. Next, we explain how a lower bound on the number of time points can be obtained. Finally, we describe in more detail how trees and nodes are solved.

3.5.1 Branching Rule

Consider an arbitrary stack s and time point t. For this combination of stack and time point three actions are possible. Either (a) a container is added to the stack (called Add), (b) a container is removed from the stack (Remove), or (c) no move is performed (Nothing). When applying branching, one of these three actions is forced for stack s at time t. For instance, if Add is forced, only sequences that add a container (of any priority level) to stack s at time t should still be considered.

Observe that this branching rule can be easily incorporated into the separation oracle described in Section 3.4.2. Suppose that Add is forced for stack s at time t. Then, all intervals for stack s which end at time t are deleted, i. e., all intervals for which a container is removed at time t are deleted. Furthermore, a value of $2B$ (see Section 3.4.2.1) is added to the weight of all intervals for stack s that start at time t, turning these intervals into heavy intervals. By the same reasoning as in the proof for Lemma 3.17, this enforces that exactly one of these intervals will be selected. Similar steps are taken if Remove is forced for stack s at time t. Now, all intervals which start at time t are deleted, and all intervals that end at time t get an additional weight of $2B$. Finally, if the action Nothing is forced for stack s at time t, then all intervals that either start or end at time t are deleted. Note that at most one container is moved per time point. Hence, if Add or Remove is forced for stack s at time t, then this action is no longer feasible for any other stack at time t. In this case, all intervals that start (in case of Add) or end (in case of Remove) at time t can be deleted for all other stacks. Clearly, none of these steps increase the difficulty of the separation oracle.

The stack and time point on which are branched, are determined as follows. Let t^* denote the minimum time point such that either Add or Remove, or both, is not forced for any stack. As adding or removing a container can be forced for at most one stack per time point, this implies that for each time point in the range $1, \ldots, t^*-1$, adding or removing a container is forced for exactly one stack. Hence, for these time points the exact move is fixed (and thus known). Hence, t^* is the minimum time point for which the exact move is not yet fixed. We consider all stacks for which no action is forced at time t^*. From these stacks we take the stack s for which the sum of $x_{s,L}$ variables, such that for sequence L

either a container is added or removed at time t^*, is maximized. More formally, we take the stack s for which $\sum_{L \in \mathcal{L}_s} \sum_{\ell \in [k]} (\mathsf{add}(L, \ell, t^*) + \mathsf{rem}(L, \ell, t^*)) x_{s,L}$ is maximized. In case of a tie, we take the stack s that was considered first.

3.5.2 Lower Bound

We say that a container is *wrongly placed* if it is positioned on top of a container that (a) has a higher priority, or (b) is itself wrongly placed. The *mis-overlay index of a stack* is defined as the number of wrongly placed containers in that stack. Similarly, the *mis-overlay index of a lay-out* is equal to the total number of wrongly placed containers. Note that this definition corresponds to the one used by Lee and Chao [55]. Clearly, all wrongly placed containers need to be moved to obtain a target lay-out, i. e., the number of moves is at least the mis-overlay index of the initial lay-out. However, if all stacks contain a wrongly placed container, then moving one does not decrease the number of wrongly placed containers. The number of wrongly placed containers can only be decreased if at least one stack contains no wrongly placed containers, i. e., if at least one stack has a mis-overlay index of zero. The minimum number of moves required to obtain a stack without wrongly placed containers is equal to the lowest mis-overlay index over all stacks. Hence, as lower bound on the number of moves we take the mis-overlay index of the initial lay-out plus the minimum mis-overlay index over all stacks.

3.5.3 Solving Trees

For each node in a tree, the same number of time points, say T, are available to move containers. Suppose that in the current tree no feasible (integer) solution is found. Hence, a solution with at most T moves does not exist. In this case, the number of allowed moves T is increased with one, and a new tree is considered. If, however, a feasible (integer) solution is found, we claim that it is optimal. Even more, it will consist of exactly T moves.

To see why this is true, observe the following. The feasible solution that is found contains at most T moves. If it is found in the first considered tree, then the lower bound on the number of moves is also equal to T, which indicates that the current feasible solution is an optimal one containing exactly T moves. If the

tree in which the feasible solution is found is not the first considered tree, then the previous considered tree contained $T-1$ time points. In this tree no feasible solution was found, so that we know that a feasible solution with at most $T-1$ moves does not exist. This immediately implies that the feasible solution is also optimal, and contains exactly T moves.

For exploring the tree we apply a depth first search. The node for which t^* (see Section 3.5.1) is maximized, is considered next. In case of a tie, we take the most recently generated node.

3.5.4 Solving Nodes

Assume that for the currently considered tree T time points are available. For each node we start with an ILP model that contains (if any) previously generated sequences, and for each stack a dummy sequence. Note that only sequences that adhere to previous branching decisions are included. Hence, if Add is forced for stack s at time t, then only sequences that add a container to stack s at time t are considered. The dummy sequences are added to ensure that a feasible solution for the LP relaxation always exists. They perform no moves, and have a cost of $T+1$. This initial model is solved, and as long as there are sequences with negative reduced cost, they are added and the model is solved again. Note that at each iteration at most one sequence per stack is added.

If there are no more sequences with negative reduced cost, we consider the LP value. If the LP value strictly exceeds T, we discard the node. Any integral solution found in a sub-tree of this node will have a cost of at least $T+1$. As only one move per time point is allowed, and we have T time points, this implies that at least one of the dummy sequences is selected. Hence, any integral solution found in the sub-tree of this node will not be feasible. If the LP value does not exceed T, we check if the solution is integral. If it is integral, we have found an optimal solution, and we stop the solve procedure. Otherwise, we apply the branching rule and continue with the next node.

To keep the number of sequences stored in memory under control, the pool of sequences is cleaned every 100 nodes. All sequences that have not been used

Algorithm 1 Branch-and-Price algorithm

```
1  procedure PREMARSHAL
2    set T equal to the lower bound for the number of moves
3    while optimal solution not found do
4      start with tree consisting of only a root node
5      while exist unpruned leaf node do
6        N := leaf node for which t* is maximized
7        initialize LP model with dummy and "valid" sequences
8        solve LP model with column generation
9        update sequence pool
10       if LP value > T then
11         prune node N
12       else if solution integral then
13         output optimal solution and prune all nodes
14       else
15         let (s, t) denote the stack and time to branch on, add children
             N1/N2/N3 := N with Add / Remove / Nothing fixed for (s, t)
16       end if
17       every 100 nodes clean sequence pool
18     end while
19     T := T + 1
20   end while
21 end procedure
```

since the last cleanup, i. e., whose corresponding variable had value zero in the solution for all LP models since the last cleanup, are discarded.

3.6 EXPERIMENTAL RESULTS

In this section we evaluate the branch-and-price algorithm described in Section 3.5. We impose a time limit of one hour for each instance, and we only consider results for PRIORITY STACKING.

3.6.1 Setup

The algorithm is implemented in C++ in combination with CPLEX 12.6, run on a machine with an Intel Core 2 Duo E8400 3.00 GHz processor and 4 GB RAM, and evaluated on randomly generated instances. To the best of our knowledge no library with real-life instances for the premarshalling problem exists, and randomly generated instances are also used in for instance [11] and [16].

The instances depend on four input parameters: the number of priorities, the number of stacks, the height of the stacks, and the fill grade, i. e., the percentage of available places that is occupied. For possible values for the number of priorities we consider [56]. To the best of our knowledge, this is the only other paper that evaluates an exact algorithm for the premarshalling problem. The authors basically consider two instances, with 3 and 6 priority levels, respectively. Therefore we consider 2 (the minimum value possible), 3 and 6 priority levels. For the other parameters, Lee and Chao [55] observe that 12 stacks with a height of 6 is already larger than most equipment can handle, and a fill grade of 75% is considered moderately high. A minimum height of 4 is observed in general. As we apply an exact method, we consider slightly lower parameter values. For the number of stacks we take values 3, 5, 7, and 9, for the height we consider 4 and 6 containers, and for the fill grade we take either 50% or 70%. We consider a fill grade of 50% as low, and a fill grade of 70% as average.

For an overview of the possible values for these parameters, which are called *Priorities*, *Stacks*, *Height*, and *Fill*, respectively, see Table 15. The number of containers is determined by multiplying the number of available positions, i. e., Stacks times Height, with Fill. In case this number is fractional, it is truncated.

Priorities	Stacks	Height	Fill (%)
2, 3, 6	3, 5, 7, 9	4, 6	50, 70

Table 15: Possible parameter values

Using these possible parameter values, we construct the instances as follows. First, consider the case where Priorities has value 6. Consider the containers one by one, and consecutively assign them priority 1 to 6. This implies that there is at most one more container with priority 1 than with priority 6. The initial lay-out is determined by placing a randomly picked container on a randomly selected non-full stack, until all containers are placed.

Second, consider the case where Priorities has value 2 or 3. In this case, the instances are based on the ones where Priorities has value 6. Each stack has the same number of containers, but the priorities of the containers are updated. For Priorities equal to 2, the three lowest and the three highest priorities are grouped together. Hence, containers which initially had priority 1, 2, or 3 will now get priority 1, while containers with initial priority 4, 5, or 6 obtain priority 2. For Priorities equal to 3 the lowest two, middle two, and highest two priorities are grouped together.

If for any of the three values for Priorities the instance does not contain any wrongly placed containers, i. e., no premarshalling operations are necessary, all three instances are discarded. This procedure is repeated until 20 instances are generated for all 48 combinations of parameter values, resulting in 960 instances.

3.6.2 Results

Over all 960 instances, the average number of containers is 17.81, of which on average 7.13 are initially wrongly placed (see Section 3.5.2). This results in an average mis-overlay index of 38.3%. The higher the value for Priorities, Fill, and especially Height, the higher the mis-overlay index. For Stacks the value of the mis-overlay index is fairly constant. This follows from the way the instances are generated: the number of containers is (almost) linear in the number of stacks, and for each container a random stack is selected.

Out of all the instances, 945 (98.4%) are solved within one hour, 895 (93.2%) within one minute, and 680 (70.8%) within one second. The 15 instances that are not solved within one hour all have value 6 for Priorities, value 6 for Height, and value 70 for Fill. The number of unsolved instances is 1, 4, 4, and 6 for 3, 5, 7, and 9 stacks, respectively.

For the 945 solved instances 8.43 moves are performed on average, which is 1.41 moves above the lower bound of 7.02 moves. On average 0.48 moves are performed per container, and 1.21 per wrongly placed container. For the 15 unsolved instances we of course do not know the number of moves, but we can use the number of time points in the tree being considered at the one hour time limit as lower bound. At least 22.53 moves are performed on average, 4.06 above the lower bound of 18.47. While the average number of moves per container is significantly higher (0.78 versus 0.48), there is only a small difference in the number of moves per wrongly placed container (1.25 versus 1.21). If we consider the results for Priorities, Height, and Fill, we observe an increase in the number of moves, the number of moves per container, and the number of moves per wrongly placed container for higher parameter values. For Stacks we observe an increasing value for the number of moves, and a decreasing value for the number of moves per container c.q. per wrongly placed container. The reason for this decreasing trend is that with more stacks there are more options to temporarily store containers, so that wrongly placed containers are re-handled less often. Observe that for Stacks equal to 3 the number of moves per container is 0.70 for the solved instances and 1.58 for the single unsolved instance, indicating that this instance has a particularly difficult to solve lay-out.

For the solved instances the integrality gap is on average 1.13 and maximally 2.80. For this problem we define the integrality gap as the ratio between the optimal (integer) solution and the lowest observed LP relaxation value. The lowest LP value is obtained at the root node of the tree that contains the optimal solution. This follows from the fact that adding forced actions increases the LP value, and adding time points decreases the LP value. For the unsolved instances we cannot determine the integrality gap, but we can give a lower bound by taking the number of time points and the value of the LP relaxation at the root node of the last considered tree. For the unsolved instances the lower bound on the average and maximum integrality gap is 1.18 and 1.85, respectively. Although

this value is biased, there does not appear to be a big difference in integrality gap between the solved and unsolved instances. In fact, the integrality gap for instances with a running time between one minute and one hour (1.32) is higher than the integrality gap for the unsolved instances. Furthermore, the maximum observed integrality gap is decreasing for longer running times. Also noteworthy is the decreasing trend of the average and maximum integrality gap for Stacks, from 1.34 and 2.80 for Stacks equal to 3, down to 1.03 and 1.18 for Stacks equal to 9.

On average 2.41 trees are solved for the solved instances. Out of these trees, on average 0.80 are *killed immediately*. A tree is killed immediately if the LP value of the root node exceeds the number of time points. As only the root node is solved for killed trees, solving these trees generally requires little time. Hence, on average 1.61 trees are *actually* solved per instance. For the unsolved instances the number of trees that need to be solved is not known, but the number of trees that was considered before the time limit can be used as a lower bound. The average number of solved, killed, and actually solved trees for the unsolved instances is 5.07, 2.13, and 2.93, respectively. For Priorities, Height, and Fill the average number of actually solved trees is increasing, while for Stacks it is decreasing.

The average number of solved nodes over all instances is 295.7. Especially for Priorities equal to 6 there is a huge increase in the number of solved nodes (779.1, compared to 35.2 and 72.7 for Priorities equal to 2 and 3, respectively). The average and maximum number of nodes in memory is 4.78 and 56, respectively. The average number of nodes in memory is increasing in all four parameters. For the unsolved instances the average number of nodes in memory rises to 15.50, which is low compared to the 8301.2 solved nodes. The maximum number of nodes in memory is obtained for a solved instance, for the unsolved instances the maximum is 39. Hence, memory usage with respect to the number of nodes appears to be low and fairly stable over time, i. e., longer running times do not lead to a huge increase in the number of nodes in memory.

The average and maximum number of generated sequences is 14,396 and 1,491,758, respectively. However, since every 100 nodes the sequence pool is cleaned and unused sequences are discarded, at most 59,729 sequences are stored in memory. In this case, the maximum is obtained by an unsolved instance, but the

maximum over the solved instances is 56,554. A surprising result is that most sequences are generated (and also stored in memory) for Stacks equal to 3. The number of generated sequences is roughly twice as high as for the other values (22,386, compared to 11,218, 11,107, and 12,875). Since each sequence only applies to a single stack, the number of generated sequences per stack is decreasing with respect to the number of stacks. Because unused sequences are discarded, the number of sequences stored in memory are kept at acceptable levels, at the expense of possibly generating the same sequence multiple times. Observe that memory usage can be lowered by cleaning the sequence pool more often.

The average running time is 93.65 seconds. The main contributors are solving the LP relaxation (46.66 seconds, or 49.8%), generating columns (34.51 seconds, 36.8%), and clearing the CPLEX models (11.67 seconds, 12.5%). This leaves only 0.80 seconds (0.9%) of overhead. If we compare the solved and unsolved instances, we observe little difference in the relative amount spend on solving the LP relaxation (47.5% versus 51.4%), generating columns (40.0% versus 34.7%), and clearing the CPLEX model (11.5% versus 13.1%). The average running time is increasing in all four parameters. For Priorities, Height, and Fill there is a big increase. Part of this increase can be contributed to the unsolved instances, which have a running time of roughly 3600 seconds. Still, if we only consider the solved instances, we observe a substantial difference. It is also noteworthy that for the solved instances, the lowest running time is observed for Stacks equal to 5 (19.57 seconds). For Stacks equal to 3 the running time is 30.15 seconds, roughly 1.5 times as high. This difference primarily stems from the time spend on generating columns (6.21 seconds versus 16.13 seconds). The reason for this difference is unknown.

On average 0.32 seconds are needed to solve one node. The average time is higher for the unsolved instances compared to the solved instances (0.43 versus 0.23 seconds). The time needed per node is strongly increasing in Priorities (from 0.02 and 0.07 to 0.35 seconds), Height (from 0.04 to 0.32 seconds), and Fill (from 0.04 to 0.35 seconds). Somewhat surprising is the fact that the highest time is obtained for Stacks equal to 5 (0.44 seconds), roughly 1.5 times higher than for Stacks equal to 3, 7, and 9 (0.26, 0.30, and 0.30 seconds, respectively). If we only consider the solved instances, we do not observe substantial differences, with

values ranging from 0.19 seconds (for Stacks equal to 7) to 0.26 seconds (for Stacks equal to 9).

An overview of the results can be found in Appendix 3.A.

3.7 CONCLUSION

We considered the intra-bay premarshalling problem. The objective is to transform the initial lay-out into a target lay-out in the minimum number of moves possible. We showed that the premarshalling problem is NP-hard, even for a fixed stack height of at least six. We developed an exact algorithm based on branch-and-price, that is evaluated on 960 randomly generated instances. For PRIORITY STACKING, 945 instances are solved within one hour, 895 within one minute, and 680 within one second. Preliminary experiments show that the algorithm runs much faster for CONFIGURATION STACKING.

An interesting topic for future research concerns the solution approach. Currently, either the optimal solution is found, or no solution is found at all. By for instance incrementing the number of time points T (see Section 3.5) with more than one, it might be possible to look for other feasible (near-optimal) solutions. Another option would be to allow multiple moves per time point. A restriction would be that each stack is involved in at most one move per time point. This decreases the number of time points, which hopefully results in a reduced running time.

3.A OVERVIEW OF RESULTS

Tables 16, 17, 18, 19, 20, and 21 contain an overview of the results. For Table 16 the instances are split according to their running time. The first column contains the results for all instances and the second column for all instances that are solved to optimality. Columns three through six contain the results for instances with a running time of less than one second, between one second and one minute, between one minute and one hour, and more than one hour, respectively. Note that the instances with a running time of more than one hour are exactly the instances that are not solved to optimality. For Tables 17, 18, 19, 20, and 21 the instances are split into three groups, namely all instances, instances solved to optimality, and instances that are not solved within one hour. Within

each group, the instances are split according to the possible parameter values. Table 17 contains the results for Priorities, Tables 18 and 19 for Stacks, Table 20 for Height, and Table 21 for Fill.

statistic	all	optimal	< 1 sec	1 sec - 1 min	1 min - 1 hour	> 1 hour
# instances	960	945	680	215	50	15
avg containers	17.81	17.64	15.34	23.33	24.44	28.93
avg mis-overlay (#)	7.13	6.96	4.91	11.73	14.20	18.07
avg mis-overlay (%)	38.3	37.9	32.3	50.7	58.9	63.4
avg lower bound	7.20	7.02	4.93	11.87	14.68	18.47
avg moves	8.65	8.43	5.83	14.27	18.80	22.53
avg moves/cont.	0.49	0.48	0.38	0.61	0.77	0.78
avg moves/w. p. cont.	1.21	1.21	1.19	1.22	1.32	1.25
avg integrality gap	1.13	1.13	1.09	1.19	1.32	1.18
max integrality gap	2.80	2.80	2.80	2.43	2.09	1.85
avg solved trees	2.45	2.41	1.90	3.40	5.12	5.07
avg killed trees	0.82	0.80	0.54	1.41	1.72	2.13
avg actual trees	1.63	1.61	1.36	1.99	3.40	2.93
avg nodes solved	295.7	168.6	8.4	117.0	2569.2	8301.2
avg nodes memory	4.78	4.61	2.83	8.67	11.30	15.50
max nodes memory	56	56	20	39	56	39
avg seq. generated	14,396	8,067	271	4,240	130,542	413,156
max seq. generated	1,491,758	717,517	1,638	40,976	717,517	1,491,758
max seq. memory	59,729	56,554	1,641	19,536	56,554	59,729
avg run time (sec)	93.65	37.98	0.23	8.42	678.39	3600.84
avg lp time (sec)	46.66	18.03	0.05	3.02	327.05	1850.74
avg gen. time (sec)	34.51	15.20	0.10	4.60	266.16	1250.63
avg clear time (sec)	11.67	4.36	0.00	0.60	79.81	472.30
avg time/node (sec)	0.32	0.23	0.03	0.07	0.26	0.43

Table 16: Overview of results split on time

statistic	all			optimal			> 1 hour		
	P = 2	P = 3	P = 6	P = 2	P = 3	P = 6	P = 2	P = 3	P = 6
# instances	320	320	320	320	320	305	0	0	15
avg containers	17.81	17.81	17.81	17.81	17.81	17.27	-	-	28.93
avg mis-overlay (#)	5.97	7.18	8.24	5.97	7.18	7.76	-	-	18.07
avg mis-overlay (%)	32.3	38.4	44.1	32.3	38.4	43.2	-	-	63.4
avg lower bound	5.99	7.25	8.36	5.99	7.25	7.86	-	-	18.47
avg moves	6.93	8.63	10.41	6.93	8.63	9.81	-	-	22.53
avg moves/cont.	0.39	0.48	0.58	0.39	0.48	0.57	-	-	0.78
avg moves/w. p. cont.	1.16	1.20	1.26	1.16	1.20	1.26	-	-	1.25
avg integrality gap	1.05	1.13	1.20	1.05	1.13	1.21	-	-	1.18
max integrality gap	1.75	2.40	2.80	1.75	2.40	2.80	-	-	1.85
avg solved trees	1.93	2.38	3.05	1.93	2.38	2.95	-	-	5.07
avg killed trees	0.67	0.78	1.01	0.67	0.78	0.96	-	-	2.13
avg actual trees	1.26	1.60	2.03	1.26	1.60	1.99	-	-	2.93
avg nodes solved	35.2	72.7	779.1	35.2	72.7	409.1	-	-	8301.2
avg nodes memory	3.43	4.62	6.28	3.43	4.62	5.83	-	-	15.50
max nodes memory	36	50	56	36	50	56	-	-	39
avg seq. generated	481	2,580	40,128	481	2,580	21,783	-	-	413,156
max seq. generated	38,335	148,267	1,491,758	38,335	148,267	717,517	-	-	1,491,758
max seq. memory	4,979	26,911	59,729	4,979	26,911	56,554	-	-	59,729
avg run time (sec)	0.72	5.04	275.18	0.72	5.04	111.62	-	-	3600.84
avg lp time (sec)	0.24	2.07	137.69	0.24	2.07	53.44	-	-	1850.74
avg gen. time (sec)	0.31	2.41	100.80	0.31	2.41	44.25	-	-	1250.63
avg clear time (sec)	0.05	0.39	34.59	0.05	0.39	13.06	-	-	472.30
avg time/node (sec)	0.02	0.07	0.35	0.02	0.07	0.27	-	-	0.43

Table 17: Overview of results for parameter Priorities

statistic	all		optimal		> 1 hour	
	S = 3	S = 5	S = 3	S = 5	S = 3	S = 5
# instances	240	240	239	236	1	4
avg containers	8.75	15.00	8.74	14.90	12.00	21.00
avg mis-overlay (#)	3.53	6.22	3.51	6.09	9.00	13.75
avg mis-overlay (%)	39.4	39.5	39.2	39.0	75.0	65.5
avg lower bound	3.71	6.30	3.68	6.16	11.00	14.25
avg moves	6.21	7.63	6.15	7.43	19.00	19.50
avg moves/cont.	0.71	0.51	0.70	0.50	1.58	0.93
avg moves/w. p. cont.	1.76	1.23	1.75	1.22	2.11	1.42
avg integrality gap	1.34	1.09	1.34	1.09	1.85	1.27
max integrality gap	2.80	1.50	2.80	1.50	1.85	1.37
avg solved trees	3.50	2.34	3.48	2.27	9.00	6.25
avg killed trees	1.00	0.75	1.00	0.72	2.00	2.25
avg actual trees	2.50	1.59	2.48	1.55	7.00	4.00
avg nodes solved	172.4	181.6	142.1	84.9	7419.0	5883.8
avg nodes memory	2.14	3.90	2.12	3.78	6.16	11.00
max nodes memory	14	30	14	28	13	30
avg seq. generated	22,386	11,218	16,238	3,846	1,491,758	446,171
max seq. generated	1,491,758	592,369	717,517	225,518	1,491,758	592,369
max seq. memory	59,729	30,759	56,554	24,376	59,729	30,759
avg run time (sec)	45.03	79.26	30.15	19.57	3600.40	3600.74
avg lp time (sec)	18.55	44.37	12.26	10.77	1522.24	2026.50
avg gen. time (sec)	24.14	24.43	16.13	6.21	1937.23	1099.05
avg clear time (sec)	2.08	10.17	1.56	2.44	125.85	466.12
avg time/node (sec)	0.26	0.44	0.21	0.23	0.49	0.61

Table 18: Overview of results for parameter Stacks - values 3 & 5

statistic	all		optimal		> 1 hour	
	S = 7	S = 9	S = 7	S = 9	S = 7	S = 9
# instances	240	240	236	234	4	6
avg containers	20.75	26.75	20.61	26.49	29.00	37.00
avg mis-overlay (#)	7.85	10.91	7.68	10.62	18.25	22.33
avg mis-overlay (%)	35.4	38.8	34.9	38.3	62.9	60.4
avg lower bound	7.89	10.91	7.70	10.62	18.75	22.33
avg moves	8.88	11.90	8.64	11.56	22.75	25.00
avg moves/cont.	0.43	0.44	0.42	0.44	0.78	0.68
avg moves/w. p. cont.	1.13	1.09	1.13	1.09	1.25	1.12
avg integrality gap	1.05	1.03	1.05	1.03	1.12	1.05
max integrality gap	1.33	1.18	1.33	1.18	1.16	1.08
avg solved trees	1.99	1.99	1.94	1.95	5.00	3.67
avg killed trees	0.70	0.83	0.67	0.81	2.50	1.83
avg actual trees	1.28	1.16	1.26	1.14	2.50	1.83
avg nodes solved	330.5	498.3	203.1	245.3	7842.5	10365.7
avg nodes memory	5.40	7.67	5.20	7.38	17.43	18.77
max nodes memory	39	56	38	56	39	37
avg seq. generated	11,107	12,875	5,855	6,209	320,946	272,852
max seq. generated	474,298	591,631	355,132	250,508	474,298	591,631
max seq. memory	19,301	22,237	15,563	22,237	19,301	20,217
avg run time (sec)	98.88	151.41	39.52	62.97	3601.50	3600.54
avg lp time (sec)	52.15	71.59	19.77	29.49	1962.94	1713.51
avg gen. time (sec)	31.36	58.10	13.64	24.89	1076.63	1353.24
avg clear time (sec)	14.60	19.85	5.65	7.87	542.75	487.18
avg time/node (sec)	0.30	0.30	0.19	0.26	0.46	0.35

Table 19: Overview of results for parameter Stacks - values 7 & 9

statistic	all		optimal		> 1 hour	
	H = 4	H = 6	H = 4	H = 6	H = 4	H = 6
# instances	480	480	480	465	0	15
avg containers	14.25	21.38	14.25	21.13	-	28.93
avg mis-overlay (#)	4.47	9.79	4.47	9.52	-	18.07
avg mis-overlay (%)	31.4	45.1	31.4	44.5	-	63.4
avg lower bound	4.47	9.93	4.47	9.65	-	18.47
avg moves	5.34	11.97	5.34	11.63	-	22.53
avg moves/cont.	0.37	0.56	0.37	0.55	-	0.78
avg moves/w. p. cont.	1.19	1.22	1.19	1.22	-	1.25
avg integrality gap	1.09	1.16	1.09	1.16	-	1.18
max integrality gap	2.80	2.35	2.80	2.35	-	1.85
avg solved trees	1.86	3.05	1.86	2.98	-	5.07
avg killed trees	0.50	1.15	0.50	1.12	-	2.13
avg actual trees	1.37	1.90	1.37	1.86	-	2.93
avg nodes solved	13.6	577.8	13.6	328.7	-	8301.2
avg nodes memory	2.76	6.79	2.76	6.51	-	15.50
max nodes memory	28	56	28	56	-	39
avg seq. generated	543	28,250	543	15,833	-	413,156
max seq. generated	68,279	1,491,758	68,279	717,517	-	1,491,758
max seq. memory	20,704	59,729	20,704	56,554	-	59,729
avg run time (sec)	0.59	186.70	0.59	76.57	-	3600.84
avg lp time (sec)	0.20	93.12	0.20	36.43	-	1850.74
avg gen. time (sec)	0.27	68.74	0.27	30.62	-	1250.63
avg clear time (sec)	0.04	23.31	0.04	8.82	-	472.30
avg time/node (sec)	0.04	0.32	0.04	0.23	-	0.43

Table 20: Overview of results for parameter Height

statistic	all		optimal		> 1 hour	
	F = 50	F = 70	F = 50	F = 70	F = 50	F = 70
# instances	480	480	480	465	0	15
avg containers	15.00	20.63	15.00	20.36	-	28.93
avg mis-overlay (#)	5.40	8.86	5.40	8.56	-	18.07
avg mis-overlay (%)	35.5	41.1	35.5	40.4	-	63.4
avg lower bound	5.43	8.97	5.43	8.66	-	18.47
avg moves	6.19	11.12	6.19	10.75	-	22.53
avg moves/cont.	0.41	0.54	0.41	0.53	-	0.78
avg moves/w. p. cont.	1.15	1.26	1.15	1.26	-	1.25
avg integrality gap	1.08	1.17	1.08	1.17	-	1.18
max integrality gap	1.93	2.80	1.93	2.80	-	1.85
avg solved trees	1.76	3.15	1.76	3.08	-	5.07
avg killed trees	0.42	1.22	0.42	1.19	-	2.13
avg actual trees	1.34	1.93	1.34	1.89	-	2.93
avg nodes solved	57.2	534.2	57.2	283.7	-	8301.2
avg nodes memory	3.30	6.25	3.30	5.95	-	15.50
max nodes memory	31	56	31	56	-	39
avg seq. generated	1,152	27,641	1,152	15,205	-	413,156
max seq. generated	66,710	1,491,758	66,710	717,517	-	1,491,758
max seq. memory	19,536	59,729	19,536	56,554	-	59,729
avg run time (sec)	2.38	184.92	2.38	74.72	-	3600.84
avg lp time (sec)	0.77	92.56	0.77	35.85	-	1850.74
avg gen. time (sec)	1.24	67.78	1.24	29.62	-	1250.63
avg clear time (sec)	0.20	23.15	0.20	8.66	-	472.30
avg time/node (sec)	0.04	0.35	0.04	0.26	-	0.43

Table 21: Overview of results for parameter Fill

4

SUBCONTRACTOR SCHEDULING

4.1 INTRODUCTION

Subcontractors help companies with projects by offering external resources and expertise. This way the company can reduce the completion time of the project and save cost. Subcontracting is a common practice in industries that face volatile demand and hard capacity constraints. When multiple projects can benefit from the use of a subcontractor, it also becomes a question of how to schedule the subcontractor over the projects. Further complications arise from the fact that such projects are usually run by different divisions or teams whose interest is to maximize their *own* savings. Therefore the company is facing an *optimization* and a *coordination* problem at the same time. To illustrate the importance of this topic we recall the case of Boeing's Dreamliner supply chain, where the lack of attention to these matters resulted in overloaded schedules of subcontractors and long delays in the overall production (see [73] for further details and references).

We model this problem as follows. There are multiple jobs endowed with a dedicated machine, where each job is characterized by a release date, processing time, and weight that represents opportunity costs. We assume there is only a single subcontractor that assists jobs in reducing their completion time, by temporarily speeding up their operation by a factor two. The subcontractor can only assist one job at a time. Given such a setting our goal is to find a schedule for the subcontractor, that indicates for each time point which job to assist, which maximizes the weighted sum of savings. To facilitate the understanding of the problem, a small instance is visualized in Figure 10.

In our context minimizing the weighted sum of completion times is equivalent to maximizing the weighted sum of savings. The reason for working with the latter

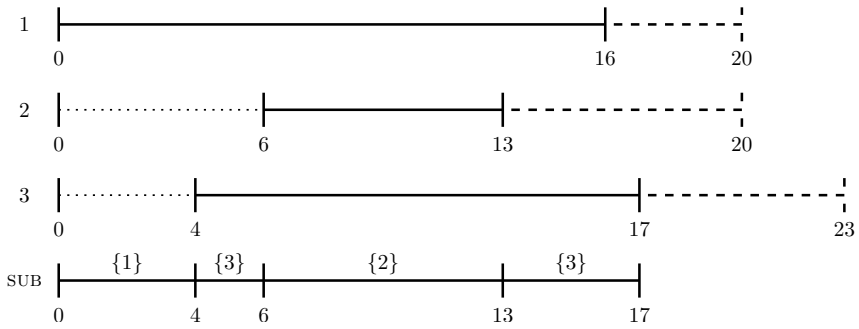

Figure 10: Small example of a feasible schedule for three jobs. The release dates are 0, 6, and 4, the processing times 20, 14, and 19, and the weights 5, 10, and 8, respectively. The first three schemes, called 1 to 3, depict the schedules for the dedicated machines. The fourth scheme, called SUB, depicts the schedule for the subcontractor machine. The dotted lines represent idle time, indicating that no job is currently available for that machine. The solid lines indicate the times on which the machines are busy. For SUB also the job that is assisted is depicted. The dashed lines indicate the additional time required on the dedicated machines, if the subcontractor would not have been employed. The time saving obtained by the jobs is 4, 7, and 6, respectively. The weighted sum of savings is 138.

objective is the economic interpretation. Companies are interested in the monetary value of the time they save by subcontracting. As the optimal solution is the same for both objectives, we can also relate to the vast amount of research on the minimization of the weighted sum of completion times in different settings.

We study this problem purely from an algorithmic viewpoint. We are interested in developing and evaluating algorithms that construct feasible, maybe even optimal, solutions, taking all input as given. For a restricted setting of the problem, without release dates and weights, it is a known result that scheduling the jobs in non-decreasing order of their processing time gives the optimal solution. We extend upon this result by showing that this order is still optimal if release dates are added. If we furthermore add weights, then ordering the jobs in non-decreasing order of their processing time is no longer optimal. Therefore, we consider additional heuristics that besides the processing time also take the weight into account when determining the schedule.

4.1.1 Related Literature

For the problem considered in this chapter, jobs are not only processed on a dedicated machine, but can also be outsourced to a third party. Hence, jobs can be processed in parallel on two machines. This type of problem resembles infinitely divisible job-shops (see e. g., [3]). Practical examples of such production types include parallel and distributed computer systems (see, e. g., [8, 26]). In these applications computations are performed by private computers, but additional computation power can be acquired via shared servers.

In the pioneering work of [67] it was shown that in a single-stage production, the sum of completion times is minimized by processing the jobs in non-decreasing order of their processing times, known as the shortest processing time first (SPT) rule. Furthermore, if every job has a positive weight that represents its relative importance, the weighted sum of completion time is minimized by ordering the jobs in non-increasing order of ratio between weight and processing time, known as Smith's rule. Unfortunately, by introducing release dates the problem becomes NP-hard, even if we allow for preemption [54]. Bruno, Coffman Jr., and Sethi [13] show that if jobs can be processed on parallel machines, minimizing the weighted sum of completion times is already NP-hard without release dates.

There is a strong link between the subcontractor scheduling problem and scheduling jobs whose processing time is dependent on the time that they are processed. In the latter problem the processing time of jobs can change over time. One can relate the two by setting for each job the starting time to half the original processing time, and the decay rate to one. Bachman, Cheng, Janiak, and Ng [5] prove that if the decay rates can differ per job, then minimizing the weighted sum of completion times is NP-hard. For a concise review of the related literature see [17].

A restricted version of our problem, with no release dates and equal weights, is considered in multiple papers. The performance of an optimal centralized schedule versus the result of a decentralized subcontracting game under different protocols is analyzed in [73, 74]. Our problem coincides with the "overlapping" protocol. The name refers to the fact that jobs can be processed simultaneously on their own machine and the subcontractor. In particular, [74] shows

that the efficient schedule orders jobs according to SPT and processes them non-preemptively.

4.1.2 Our Contributions

We consider several heuristics, in particular we show that scheduling in non-increasing order of their weights is a $2/3$ approximation, while ordering according to Smith's ratio surprisingly guarantees at best a $1/2$ approximation. We construct a polynomial time approximation scheme (PTAS) for the case when all release dates are equal. Finally, we evaluate an extended set of heuristics on randomly generated instances. As it turns out, scheduling jobs in non-increasing order of their weight not only gives the best results from a theoretical perspective, but also from a practical perspective.

4.1.3 Organization

In Section 4.2 we introduce notation and formally describe the subcontractor scheduling problem. In Section 4.3 we consider the situation where there are release dates, but no weights. In Section 4.4 weights are also added. In this situation scheduling the jobs in non-decreasing order of their processing times is not necessarily optimal, and three additional heuristics are introduced. In Section 4.5 we describe a polynomial time approximation scheme (PTAS) for the situation where there are no release dates. An extended set of heuristics is evaluated on randomly generated instances in Section 4.6. Finally, conclusions are drawn in Section 4.7.

4.2 PRELIMINARIES

The subcontractor scheduling problem, hereafter called SUBCONTRACTOR, is defined as follows. We are given a set of n jobs $\mathcal{N} = \{1, \ldots, n\}$, each with release date r_i, processing time p_i, and weight w_i. For each job a dedicated machine is available, complemented with a single subcontractor machine, hereafter called SUB, which can work on at most one job at a time.

An assignment of job $i \in \mathcal{N}$ to SUB is a function $a_i : \mathbb{R}_+ \to \{0, 1\}$, where a_i only has finitely many points of discontinuity. The schedule a is defined by the set of all functions $\{a_i\}_{i \in \mathcal{N}}$. A schedule is feasible if for each moment in time, at most

one job is assigned to SUB. Given schedule a, job i is completed at time $C_i(a)$ if the combined scheduled time between r_i and $C_i(a)$ on the dedicated machine and on SUB is equal to p_i, i.e., $C_i(a)$ is the unique solution to the equation $C_i(a) + \int_{r_i}^{C_i(a)} a_i(t)dt = p_i + r_i$. The savings that job i obtains from schedule a is denoted by $s_i(a)$. Note that this is exactly the time that job i spends on SUB before $C_i(a)$, i.e., $s_i(a) = \int_{r_i}^{C_i(a)} a_i(t)dt$.

The value of job i for schedule a, given input (r, p, w), is $v_i(a) = w_i s_i(a)$. The total value equals the sum of values over all jobs, i.e., $v(a) = \sum_{i\in\mathcal{N}} v_i(a)$. The goal is to find a feasible schedule that maximizes the total value:

$$\max v(a) \text{ subject to } \sum_{i\in\mathcal{N}} a_i(t) \leqslant 1, \quad \forall t \in \mathbb{R}_+.$$

If the jobs are indexed in non-decreasing order of their processing time, we say that the jobs are indexed in SPT order. The following two observations follow immediately from the definitions of C_i and s_i, and we conclude this section with a basic result from [74].

Observation 4.1. *For any schedule* a *and any job* $i \in \mathcal{N}$, $s_i(a) \leqslant \frac{1}{2}p_i$.

Observation 4.2. *For any two schedules* a *and* b *and any job* $i \in \mathcal{N}$, $s_i(a) \leqslant s_i(b)$ *if and only if* $C_i(a) \geqslant C_i(b)$.

Theorem 4.3. *[74] Assume that* $r_i = 0$ *and* $w_i = 1$ *for all jobs* $i \in \mathcal{N}$, *and let the jobs be indexed in SPT order. Then the optimal schedule, called* spt, *is non-preemptive and it schedules jobs according to the SPT order. Moreover, the total savings of job* i *equals*

$$s_i(spt) = \frac{p_i}{2} - \sum_{j=1}^{i-1} \frac{s_j(spt)}{2} = \frac{p_i}{2} - \sum_{j=1}^{i-1} \frac{p_j}{2^{i+1-j}} \quad \Rightarrow \quad \sum_{i=1}^{n} s_i(spt) = \sum_{i=1}^{n} \frac{p_i}{2^{n+1-i}}.$$

4.3 RELEASE DATES

In this section we consider the case where the jobs have release dates and processing times, but no weights. Equivalently, one can assume that all weights are equal to one. For the case with no release dates, it is a known result that scheduling the jobs in non-decreasing order of their processing time gives the optimal solution (see [74]). We show that a slightly adapted rule, called *Proctime*, gives the optimal solution if release dates are added.

Proctime works as follows. At each release date or completion time, the available job with the lowest *remaining* processing time is scheduled on SUB. In case of a tie, the job with the lowest index is chosen. A job is available if it is released but not yet completed. This implies that if a job is available, then SUB cannot be idle. Note that jobs can be preempted on SUB, but only at the release date of another job.

Theorem 4.4. *Proctime is optimal in case of release dates and equal weights, if preemption is allowed.*

Proof. Let ptm denote the schedule obtained by Proctime, and assume to the contrary that ptm is not optimal. Let opt denote the optimal schedule where the time point T of first difference between ptm and opt is maximal, i. e., $\mathsf{ptm}_i(t) = \mathsf{opt}_i(t)$ for all jobs $i \in \mathcal{N}$ and all time point $t < T$. Hence, only the instance remaining at time T is of interest, and all choices up until T can be "ignored". Therefore, without loss of generality, we consider the instance remaining at time T, and can hence assume that the time point of first difference is equal to zero.

Claim 4.5. *Neither ptm nor opt is idle at time zero.*

Proof. By definition it cannot be that both are idle. Hence, at least one job, say i, is available. If a job is available, then by definition ptm cannot be idle. Assume to the contrary that opt is idle at time zero. Hence, there is an $\epsilon > 0$ such that opt is idle on the interval $[0, \epsilon)$. If we now schedule i during this interval, we increase the value of this job, and thus the total value. This contradict that opt is an optimal schedule, so that opt can also not be idle at time zero. □

Hence, we know that at time zero ptm and opt are respectively working on some jobs i and j, with $i \neq j$ and $p_i \leqslant p_j$. Consider the feasible schedule fs, that is identical to opt for all jobs except i and j. At time zero, fs starts working on i, but only at time points where opt is working on either i or j, until i finishes for fs.

After that, fs works on j, but again only at time points where opt is working on i or j. Hence, we have

$$\mathsf{fs}_i(t) = \begin{cases} \mathsf{opt}_i(t) + \mathsf{opt}_j(t) & t \leqslant C_i(\mathsf{fs}) \\ 0 & \text{else,} \end{cases}$$

$$\mathsf{fs}_j(t) = \begin{cases} 0 & t \leqslant C_i(\mathsf{fs}) \\ \mathsf{opt}_i(t) + \mathsf{opt}_j(t) & \text{else.} \end{cases}$$

Based on the schedule fs for jobs i and j, we arrive at the following observations.

Observation 4.6. $C_i(\mathit{fs}) \leqslant C_i(\mathit{opt})$ *and* $C_i(\mathit{fs}) \leqslant C_j(\mathit{opt})$.

Observation 4.7. $C_j(\mathit{fs}) \geqslant C_i(\mathit{opt})$ *and* $C_j(\mathit{fs}) \geqslant C_j(\mathit{opt})$.

Proof. Observation 4.6 follows from the definition of fs and the fact that $p_i \leqslant p_j$. As opt is an optimal schedule and the value for all jobs except i and j are identical for fs and opt, we have by definition that $s_i(\mathsf{opt}) + s_j(\mathsf{opt}) \geqslant s_i(\mathsf{fs}) + s_j(\mathsf{fs})$, which implies that $C_i(\mathsf{fs}) + C_j(\mathsf{fs}) \geqslant C_i(\mathsf{opt}) + C_j(\mathsf{opt})$. Combined with Observation 4.6 this leads to Observation 4.7. □

Claim 4.8. *fs is an optimal schedule.*

Proof. As already observed before, fs and opt only differ on jobs i and j. We get that

$$\begin{aligned} s_i(\mathsf{opt}) + s_j(\mathsf{opt}) &= \int_0^{C_i(\mathsf{opt})} \mathsf{opt}_i(t)\,dt + \int_0^{C_j(\mathsf{opt})} \mathsf{opt}_j(t)\,dt \\ &= \int_0^{C_j(\mathsf{fs})} \mathsf{opt}_i(t) + \mathsf{opt}_j(t)\,dt \\ &= \int_0^{C_j(\mathsf{fs})} \mathsf{fs}_i(t) + \mathsf{fs}_j(t)\,dt \\ &= \int_0^{C_i(\mathsf{fs})} \mathsf{fs}_i(t)\,dt + \int_0^{C_j(\mathsf{fs})} \mathsf{fs}_j(t)\,dt \\ &= s_i(\mathsf{fs}) + s_j(\mathsf{fs}), \end{aligned}$$

where the second equality follows from Observation 4.7, and the third and fourth equalities from the definition of fs. □

Hence, fs is also an optimal schedule, which at time zero is working on job i. This contradicts that opt is the optimal schedule where the time point of first difference is maximized. Hence, ptm is an optimal schedule. □

4.4 RELEASE DATES AND WEIGHTS

In this section we consider the case where the jobs have a release date, processing time, and weight. In Section 4.3 we showed that in the situation without weights, Proctime gives the optimal solution. When adding weights this no longer holds, and as it turns out (a slightly modified version of) Proctime perform quite bad.

Therefore, we introduce three additional heuristics, that work according to the same principal as Proctime: at each release date or completion time, the available job that scores best according to some criterion is selected. Note that preemption is thus still allowed, but is only possible at the release date of another job. We consider the four heuristics in the following subsections.

4.4.1 Proctime

The definition of Proctime is slightly different from the one in Section 4.3. As first criterion still the job with the lowest remaining processing time is selected. In case of a tie, the job with the highest weight is selected. If there is still a tie, the job with the lowest index is chosen. Contrary to the case with equal weights, Proctime now performs badly.

Theorem 4.9. *Proctime is a* $\frac{1}{2^{n-1}}$*-approximation algorithm, and this ratio is tight.*

Proof. Let ptm and opt denote the schedule obtained by Proctime and the optimal schedule, respectively. Consider ptm and take an arbitrary job i. Furthermore, let $p_i^T = p_i - (T - r_i) - \int_{r_i}^{T} \mathsf{ptm}_i(t)dt$ denote the remaining processing time of job i at time $T \in [r_i, C_i(\mathsf{ptm})]$.

Conjecture 4.10. *Consider an arbitrary time point* $T \in [r_i, C_i(\mathsf{ptm})]$ *and let* C *denote the earliest completion time after* T*. Then,* $C - T \leqslant \frac{1}{2}p_i^T$*, i. e., the time between* T *and the next completion time* C *is at most half the remaining processing time of job* i *at time* T*.*

Proof. From the definition of ptm, we have that SUB is always working on the available job with the lowest remaining processing time. Hence, during an interval of length ϵ in which no job is released, the lowest remaining processing time decreases with 2ϵ, while at release dates the lowest remaining processing time can only decrease. Combining these observations, we have that after at most half of the lowest remaining processing time at T, a completion time must be encountered. Clearly, p_i^T is an upper bound for this. □

Let $k \in \{0, \ldots, n-1\}$ denote the number of completion times strictly between r_i and $C_i(\mathsf{ptm})$, and without loss of generality assume that $r_i < C_1 \leqslant \ldots \leqslant C_k < C_i(\mathsf{ptm})$. Abusing notation, we define $C_0 = r_i$ and $C_{k+1} = C_i(\mathsf{ptm})$. By Conjecture 4.10 we have for $j \in \{1, \ldots, k+1\}$ that

$$C_j - C_{j-1} \leqslant \tfrac{1}{2} p_i^{C_{j-1}} \leqslant \tfrac{1}{2}\left(p_i - C_{j-1} + r_i\right) \;\Rightarrow\; C_j \leqslant \tfrac{1}{2}\left(p_i + C_{j-1} + r_i\right).$$

Corollary 4.11. $C_j \leqslant \left(1 - \frac{1}{2^j}\right) p_i + r_i, \quad \forall j \in \{1, \ldots, k+1\}.$

Proof. We prove the corollary by induction. Thereto, first consider the case where j equals 1. We get

$$C_1 \leqslant \tfrac{1}{2}\left(p_i + C_0 + r_i\right) = \tfrac{1}{2} p_i + r_i = \left(1 - \tfrac{1}{2}\right) p_i + r_i.$$

Next, assume that the corollary holds for $1, \ldots, j$. For $j+1$ we then get

$$\begin{aligned} C_{j+1} &\leqslant \tfrac{1}{2}\left(p_i + C_j + r_i\right) \leqslant \tfrac{1}{2} p_i + \tfrac{1}{2}\left(1 - \tfrac{1}{2^j}\right) p_i + \tfrac{1}{2} r_i + \tfrac{1}{2} r_i \\ &= \left(1 - \tfrac{1}{2^{j+1}}\right) p_i + r_i, \end{aligned}$$

which shows that the corollary holds. □

Lemma 4.12. *For any* $i \in \mathcal{N}$, $s_i(ptm) \geqslant \frac{1}{2^n} p_i$.

Proof. By combining the observation that $C_{k+1} = C_i(\mathsf{ptm})$, the definition of $C_i(\mathsf{ptm})$, and Corollary 4.11, we have that

$$p_i + r_i - s_i(\mathsf{ptm}) = C_i(\mathsf{ptm}) = C_{k+1} \leqslant \left(1 - \tfrac{1}{2^{k+1}}\right) p_i + r_i \;\Rightarrow\; s_i(\mathsf{ptm}) \geqslant \tfrac{1}{2^n} p_i,$$

where the last inequality follows from the fact that $k \in \{0, \ldots, n-1\}$. □

By Observation 4.1 and Lemma 4.12 we have that

$$2^{n-1}v(\mathsf{ptm}) = 2^{n-1}\sum_{i\in\mathcal{N}} w_i s_i(\mathsf{ptm}) \geqslant \sum_{i\in\mathcal{N}} \tfrac{1}{2} w_i p_i \geqslant \sum_{i\in\mathcal{N}} w_i s_i(\mathsf{opt}) = v(\mathsf{opt}),$$

which shows that Proctime is indeed a $1/2^{n-1}$-approximation algorithm.

To see that this ratio is tight, consider the following instance. There are n jobs, split into $n-1$ "small" jobs with processing time 2^n and weight ϵ, and one "big" job with processing time $2^n+\epsilon$ and weight 1. All release dates are equal to zero. Without loss of generality, we assume that the jobs are indexed in SPT order.

First, consider the solution obtained by Proctime. Since all release dates are equal to zero, the jobs will be scheduled in non-decreasing order of their processing time. Therefore, we can apply Theorem 4.3 to obtain the time spend on SUB for the small jobs and the big job.

$$\sum_{i=1}^{n-1} s_i(\mathsf{ptm}) = \sum_{i=1}^{n-1} 2^i = 2^n - 2, \tag{21}$$

$$s_n(\mathsf{ptm}) = \tfrac{p_n}{2} - \sum_{i=1}^{n-1} \frac{s_i(\mathsf{ptm})}{2} = 2^{n-1} + \tfrac{1}{2}\epsilon - \tfrac{1}{2}\left(2^n - 2\right) = 1 + \tfrac{1}{2}\epsilon. \tag{22}$$

Combining equations (21) and (22) we get that

$$v(\mathsf{ptm}) = \sum_{i\in\mathcal{N}} w_i s_i(\mathsf{ptm}) = \left(2^n - 2\right)\epsilon + \left(1 + \tfrac{1}{2}\epsilon\right) = 1 + \left(2^n - \tfrac{3}{2}\right)\epsilon. \tag{23}$$

Second, consider the feasible schedule fs, where at time zero SUB starts working on the big job, until it is finished. We get

$$v(\mathsf{fs}) = w_n s_n(\mathsf{fs}) = 2^{n-1} + \tfrac{1}{2}\epsilon \quad \Rightarrow \quad v(\mathsf{opt}) \geqslant 2^{n-1} + \tfrac{1}{2}\epsilon. \tag{24}$$

For the ratio between ptm and opt we get from equations (23) and (24) that

$$\frac{v(\mathsf{ptm})}{v(\mathsf{opt})} \leqslant \frac{1 + \left(2^n - \frac{3}{2}\right)\epsilon}{2^{n-1} + \frac{1}{2}\epsilon} \to \frac{1}{2^{n-1}} \text{ as } \epsilon \to 0,$$

which shows that the approximation ratio is indeed tight. □

4.4.2 Weight

Whereas Proctime primarily focuses on the processing time, the heuristic *Weight* primarily focuses on the weight. More precise, Weight selects the job with the highest weight. In case of a tie, the job with the lowest remaining processing time is selected. If there is still a tie, the job with the lowest index is chosen.

Theorem 4.13. *Weight is a $\frac{2}{3}$-approximation algorithm and this ratio is tight.*

Proof. Let wht and opt denote the schedule produced by Weight and the optimal schedule, respectively. Let $\mathcal{N}^+$ denote the set of jobs that are scheduled on SUB for at least as long for wht as for opt, and let $\mathcal{N}^-$ denote its complement. More formal, $\mathcal{N}^+ := \{i \in \mathcal{N} \mid s_i(\mathsf{wht}) \geqslant s_i(\mathsf{opt})\}$ and $\mathcal{N}^- := \mathcal{N} \setminus \mathcal{N}^+$. Let $\mathcal{T}$ denote all time points t such that opt schedules some job $i \in \mathcal{N}^+$ on SUB, while wht has completed job i by time t, i.e., $\mathcal{T} := \{t \mid \exists i \in \mathcal{N}^+ : \mathsf{opt}_i(t) = 1 \text{ and } t > C_i(\mathsf{wht})\}$. Finally, define optRes as opt restricted to time points in $\mathcal{T}$, and its complement $\overline{\mathsf{optRes}}$,

$$\mathsf{optRes}_i(t) := \begin{cases} \mathsf{opt}_i(t) & \text{if } t \in \mathcal{T} \\ 0 & \text{else,} \end{cases} \qquad \overline{\mathsf{optRes}}_i(t) := \begin{cases} 0 & \text{if } t \in \mathcal{T} \\ \mathsf{opt}_i(t) & \text{else.} \end{cases}$$

Using these definitions, we arrive at the following Lemmas.

Lemma 4.14. *For any $i \in \mathcal{N}^+$, $s_i(\mathit{optRes}) \leqslant \frac{1}{2} s_i(\mathit{wht})$.*

Proof. Consider time point $t = C_i(\mathsf{wht})$. For job i the remaining processing time in optRes at time t is by definition $s_i(\mathsf{wht})$. Of this remaining time, clearly at most half can be scheduled on SUB, so that $s_i(\mathsf{optRes}) \leqslant \frac{1}{2} s_i(\mathsf{wht})$. □

Lemma 4.15. *For any time point $t \notin \mathcal{T}$, if wht is idle then opt is idle as well.*

Proof. Take an arbitrary time point $t \notin \mathcal{T}$ where wht is idle. To the contrary, assume that opt schedules some job i at time t. Since wht is idle, i is already completed for wht, so that $t > C_i(\mathsf{wht})$. As $t \notin \mathcal{T}$, this implies that $i \notin \mathcal{N}^+$, meaning that $i \in \mathcal{N}^-$. Furthermore, since opt is still working on i at time t, we have that $C_i(\mathsf{wht}) < C_i(\mathsf{opt})$, which implies by Observation 4.2 that $s_i(\mathsf{wht}) > s_i(\mathsf{opt})$. This contradicts the observation that $i \in \mathcal{N}^-$. □

Lemma 4.16. *For any time point* $t \notin \mathcal{T}$*, if* wht *processes some job* i *and* opt *processes a different job* j*, then* $w_i \geqslant w_j$.

Proof. Take an arbitrary time point $t \notin \mathcal{T}$ where wht and opt are working on different jobs i and j, respectively, and assume to the contrary that $w_i < w_j$. Since j is already released at time t, it must hold that wht has already finished j at time t. Hence, $C_j(\mathsf{wht}) < C_j(\mathsf{opt})$, so that by Observation 4.2 we have $s_j(\mathsf{wht}) > s_j(\mathsf{opt})$, and thus $j \in \mathcal{N}^+$. Since $t > C_j(\mathsf{wht})$, we have that $t \in \mathcal{T}$, which contradicts the assumption that $t \notin \mathcal{T}$. □

Lemma 4.17. $\sum_{i\in\mathcal{N}} w_i s_i(\mathit{wht}) \geqslant \sum_{i\in\mathcal{N}} w_i s_i\left(\overline{\mathit{optRes}}\right)$

Proof. From Lemma 4.15 and the definition of $\overline{\mathsf{optRes}}$ it follows that if wht is idle at time t, then $\overline{\mathsf{optRes}}$ is also idle at time t. Lemma 4.16 ensures that whenever wht is working on a job at time t, then $\overline{\mathsf{optRes}}$ is working on a job with at most the same weight. Hence, at each time point, the value obtained by wht is at least the value obtained by $\overline{\mathsf{optRes}}$. □

Note that $\sum_{i\in\mathcal{N}^+} w_i s_i(\mathsf{optRes}) = \sum_{i\in\mathcal{N}} w_i s_i(\mathsf{optRes})$ as optRes by definition does not work on jobs in $\mathcal{N}^-$. Combining the previous observations, we obtain the following

$$\begin{aligned}
\tfrac{3}{2} v(\mathsf{wht}) &= \tfrac{3}{2} \sum_{i\in\mathcal{N}} w_i s_i(\mathsf{wht}) \\
&\geqslant \tfrac{1}{2} \sum_{i\in\mathcal{N}^+} w_i s_i(\mathsf{wht}) + \sum_{i\in\mathcal{N}} w_i s_i(\mathsf{wht}) \\
&\geqslant \sum_{i\in\mathcal{N}} w_i s_i(\mathsf{optRes}) + \sum_{i\in\mathcal{N}} w_i s_i\left(\overline{\mathsf{optRes}}\right) \\
&= \sum_{i\in\mathcal{N}} w_i s_i(\mathsf{opt}) \\
&= v(\mathsf{opt}),
\end{aligned}$$

where the second inequality follows from Lemmas 4.14 and 4.17. This shows that Weight is a $2/3$-approximation algorithm.

To show that this ratio is tight, consider the following instance. There are n jobs, split into $n-1$ "light" jobs with processing time and weight 1, and one "heavy" job with processing time 2 and weight $1+\epsilon$. All release dates are zero.

Lemma 4.18. *$v(wht) = 1 + \epsilon$ and $v(opt) = \frac{3}{2} - \frac{1}{2^n} + \left(\frac{1}{2} + \frac{1}{2^n}\right)\epsilon$.*

Proof. Without loss of generality assume that the jobs are indexed in SPT order. First, consider wht, which starts with the heavy job. Hence, we get that $C_n(\mathsf{wht}) = s_n(\mathsf{wht}) = \frac{1}{2}p_n = 1$. Observe that at time 1, all other jobs also finish, so that $s_i(\mathsf{wht}) = 0$ for all $i \neq n$. Combining these observations we get that

$$v(\mathsf{wht}) = w_n s_n(\mathsf{wht}) = 1 + \epsilon.$$

Second, consider the following solution, which we claim is optimal, where the jobs are processed in SPT order, i. e., first all light jobs are processed, followed by the heavy job. Since all release dates are equal to zero, we can apply Theorem 4.3 to obtain the time spend on SUB for the light jobs and the heavy job.

$$\sum_{i=1}^{n-1} s_i(\mathsf{opt}) = \sum_{i=1}^{n-1} \tfrac{1}{2^{n-i}} = 1 - \tfrac{1}{2^{n-1}}, \tag{25}$$

$$s_n(\mathsf{opt}) = \tfrac{p_n}{2} - \sum_{i=1}^{n-1} \tfrac{s_i(\mathsf{opt})}{2} = 1 - \tfrac{1}{2}\left(1 - \tfrac{1}{2^{n-1}}\right) = \tfrac{1}{2} + \tfrac{1}{2^n}. \tag{26}$$

Combining equations (25) and (26) we get that

$$\begin{aligned} v(\mathsf{opt}) &= \sum_{i \in \mathcal{N}} w_i s_i(\mathsf{opt}) = \left(1 - \tfrac{1}{2^{n-1}}\right) + \left(\tfrac{1}{2} + \tfrac{1}{2^n}\right)(1 + \epsilon) \\ &= \tfrac{3}{2} - \tfrac{1}{2^n} + \left(\tfrac{1}{2} + \tfrac{1}{2^n}\right)\epsilon. \end{aligned}$$

□

For the ratio between wht and opt we get from Lemma 4.18 that

$$\frac{v(\mathsf{wht})}{v(\mathsf{opt})} = \frac{1 + \epsilon}{\frac{3}{2} - \frac{1}{2^n} + \left(\frac{1}{2} + \frac{1}{2^2}\right)\epsilon} \to \frac{2}{3} \text{ as } \epsilon \to 0, n \to \infty,$$

which shows that the approximation ratio is indeed tight. □

4.4.3 Smith

In the previous two sections we considered the heuristics Proctime and Weight, which primarily focus on the processing time and weight, respectively. In this

section we look at the heuristic *Smith*, which, as the name suggests, considers the well known Smith ratio. More precisely, Smith selects the job that maximizes the ratio between weight and remaining processing time. In case of a tie, the job with the highest weight is selected. If there is still a tie, the job with the lowest index is selected. Although Smith focuses both on the processing time and the weight, it performs worse than Weight.

Theorem 4.19. *Smith is at best a $\frac{1}{2}$-approximation algorithm.*

Proof. Let sth and opt denote the schedule produced by Smith and the optimal schedule, respectively. Consider the following instance, which consists of $n = 2k$ jobs, with $k \geqslant 2$. We have processing time $p_i = i + 1$ for jobs $i = 1, \ldots, k$ and $p_i = k + 2$ for jobs $k + 1, \ldots, 2k$. For all jobs we have that the weight is equal to the processing time plus ϵ, i. e., $w_i = p_i + \epsilon$, and the release date is equal to zero.

First, consider sth. Observe that initially the jobs are sorted in SPT order (see Section 4.4.1), i. e., $\frac{w_1}{p_1} \geqslant \frac{w_2}{p_2} \geqslant \ldots \geqslant \frac{w_n}{p_n}$.

Claim 4.20. *For sth, the jobs are processed in SPT order.*

Proof. We show that the order of the jobs does not change over time. Thereto, consider two jobs i and j, with $i < j$, which have not been scheduled, and let t denote an arbitrary time point between zero and p_i. At time t we have

$$\frac{w_i}{p_i - t} = \frac{p_i + \epsilon}{p_i - t} = 1 + \frac{t + \epsilon}{p_i - t} \geqslant 1 + \frac{t + \epsilon}{p_j - t} = \frac{p_j + \epsilon}{p_j - t} = \frac{w_j}{p_j - t},$$

which shows that the order of jobs does not change over time. □

Claim 4.21. $s_i(sth) = 1$ *for* $i = 1, \ldots, k$, *and* $s_i(sth) = \frac{1}{2^{i-k-1}}$ *for* $i = k + 1, \ldots, 2k$.

Proof. We prove this by induction. For the case $i = 1$ we get $s_1(\mathsf{sth}) = \frac{p_1}{2} = 1$. Now, assume the claim holds for $j = 1, \ldots, i$, and consider job $i + 1$. We get

$$s_{i+1}(\mathsf{sth}) = \frac{p_{i+1}}{2} - \sum_{j=1}^{i} \frac{s_j(\mathsf{sth})}{2} = \begin{cases} \frac{i+2}{2} - \frac{i}{2} = 1 & \text{if } i \leqslant k \\ \frac{k+2}{2} - \frac{k}{2} - \sum_{j=k+1}^{i} \frac{1}{2^{j-k}} = \frac{1}{2^{i-k}} & \text{if } i > k, \end{cases}$$

which proves the claim. □

From Claim 4.21 we get that

$$\sum_{i=k+1}^{2k} s_i(\mathsf{sth}) = \sum_{i=k+1}^{2k} \tfrac{1}{2^{i-k-1}} = 2 - \tfrac{1}{2^{k-1}}.$$

Combining these observations, we get the following for the total value of sth

$$\begin{aligned} v(\mathsf{sth}) &= \sum_{i\in\mathcal{N}} w_i s_i(\mathsf{sth}) = \sum_{i=1}^{k} (i+1+\epsilon) + \left(2 - \tfrac{1}{2^{k-1}}\right)(k+2+\epsilon) \\ &= \tfrac{k^2}{2} + \tfrac{7k}{2} - \tfrac{k+2}{2^{k-1}} + 4 + \left(k+2-\tfrac{1}{2^{k-1}}\right)\epsilon. \end{aligned} \tag{27}$$

Second, consider the following solution, which we claim is optimal, where jobs $k+1,\ldots,2k$ are processed. Observe that all these jobs have a processing time of $k+2$. Hence, these jobs are processed in non-decreasing order of their processing time. Slightly abusing Theorem 4.3 we get that

$$\sum_{i=1}^{k} s_{k+i}(\mathsf{opt}) = \sum_{i=1}^{k} \tfrac{k+2}{2^{k-i+1}} = (k+2)\left(1-\tfrac{1}{2^k}\right).$$

Observe that for $k \geqslant 2$ we have that $(k+2)\left(1-\frac{1}{2^k}\right) > k+1$, which implies that jobs $1,\ldots,k$ have already finished by the time jobs $k+1,\ldots,2k$ are processed. Hence, we get $s_i(\mathsf{opt}) = 0$ for $i = 1,\ldots,k$, so that for $v(\mathsf{opt})$ we get

$$\begin{aligned} v(\mathsf{opt}) &= \sum_{i\in\mathcal{N}} w_i s_i(\mathsf{opt}) = (k+2)\left(1-\tfrac{1}{2^k}\right)(k+2+\epsilon) \\ &\geqslant \left(k^2+4k+4\right)\left(1-\tfrac{1}{2^k}\right). \end{aligned} \tag{28}$$

For the ratio between sth and opt we get from equations (27) and (28) that

$$\frac{v(\mathsf{sth})}{v(\mathsf{opt})} \leqslant \frac{\frac{k^2}{2} + \frac{7k}{2} - \frac{k+2}{2^{k-1}} + 4 + \left(k+2-\frac{1}{2^{k-1}}\right)\epsilon}{\left(k^2+4k+4\right)\left(1-\frac{1}{2^k}\right)} \to \frac{1}{2} \text{ as } \epsilon \to 0, k \to \infty,$$

which shows that Smith is at best a $1/2$-approximation algorithm. □

4.4.4 Profit

In the previous section we considered the heuristic Smith, that primarily considers the ratio between weight and processing time. In this section we evaluate the heuristic *Profit*, which considers the product between these two inputs. More

precisely, Profit selects the job that maximizes the product of the remaining processing time and the weight. In case of a tie, the job with the highest weight is selected. If there is still a tie, the job with the lowest index is chosen.

Theorem 4.22. *Profit is a $\frac{1}{n}$-approximation algorithm and this ratio is tight.*

Proof. Let pft and opt denote the schedule obtained by Profit and the optimal schedule, respectively. Let k denote the job for which the product of processing time and weight is maximized, i. e., let k be the job such that $p_k w_k \geqslant p_i w_i$ for all $i \in \mathcal{N}$.

Consider pft, and let p_i^t denote the remaining processing time of job i at time $t \in [r_i, C_i(\mathsf{pft})]$. Outside of this range, we define p_i^t to be equal to zero. Let $\mathcal{T} = \{r_i, C_i(\mathsf{pft}) \mid i \in \mathcal{N}\}$ denote the set of all release dates and completion times. Let $v_t(\mathsf{pft})$ denote the weighted sum of savings obtained after time t. Finally, let $k(t)$ denote the job for which the product of remaining processing time and weight is maximized at time t, i. e., let $k(t)$ be the job such that $p_{k(t)}^t w_{k(t)} \geqslant p_i^t w_i$ for all $i \in \mathcal{N}$.

Lemma 4.23. *For any $t \in \mathcal{T}$, $v_t(\mathit{pft}) \geqslant \frac{1}{2} p_{k(t)}^t w_{k(t)}$.*

Proof. Observe that by definition the lemma holds for the maximum time point in $\mathcal{T}$. This time point is the completion time of the last job, which implies that the remaining processing time of all jobs is zero. Assume to the contrary that the lemma does not hold for all time points in $\mathcal{T}$. Let t denote the maximum time point for which the lemma does not hold, and let τ denote the first time point in $\mathcal{T}$ after t. By our previous observation, τ always exists. Let i and j denote the jobs for which the remaining processing time multiplied with the weight is maximized at times t and τ, respectively. Observe that between t and τ, SUB is solely working on job i, as there are no other release dates or completion times

between these two time point. Together with the definition of jobs i and j, we have that $p_j^\tau w_j \geqslant p_i^\tau w_i = (p_i^t - 2(\tau - t))w_i$. This leads to

$$\begin{aligned} v_t(\mathsf{pft}) &= (\tau - t)w_i + v_\tau(\mathsf{pft}) \\ &\geqslant (\tau - t)w_i + \tfrac{1}{2}p_j^\tau w_j \\ &\geqslant (\tau - t)w_i + \tfrac{1}{2}p_i^\tau w_i \\ &= (\tau - t)w_i + \tfrac{1}{2}\left(p_i^t - 2\,(\tau - t)\right)w_i \\ &= \tfrac{1}{2}p_i^t w_i, \end{aligned}$$

which contradicts the assumption that $v_t(\mathsf{pft}) < \frac{1}{2}p_i^t w_i$. As time point t was arbitrarily chosen, we have that the lemma holds for all $t \in \mathcal{T}$. □

Consider time point r_k, i. e., the release date of the job for which the product of processing time and weight is maximized. By definition $p_k^{r_k} = p_k$, so that

$$v(\mathsf{pft}) \geqslant v_{r_k}(\mathsf{pft}) \geqslant \tfrac{1}{2}p_k w_k. \tag{29}$$

Consider the optimal solution opt. For this solution we get

$$v(\mathsf{opt}) = \sum_{i \in \mathcal{N}} w_i s_i(\mathsf{opt}) \leqslant \sum_{i \in \mathcal{N}} \tfrac{1}{2}p_i w_i \leqslant \sum_{i \in \mathcal{N}} \tfrac{1}{2}p_k w_k = \tfrac{n}{2}p_k w_k, \tag{30}$$

where the first inequality follows from Observation 4.1.

Combining equations (29) and (30) we get that $nv(\mathsf{pft}) \geqslant \frac{n}{2}p_k w_k \geqslant v(\mathsf{opt})$, which shows that Profit is indeed a $1/n$-approximation algorithm.

To see that this ratio is tight, consider the following instance. There are n jobs, with release date $r_i = 0$, processing time $p_i = k^i$, and weight $w_i = k^{n-i}$, except for job n, which has weight $w_n = 1 + \epsilon$. Let $k \geqslant 2$. Observe that the jobs are indexed in SPT order.

First, consider pft, which starts with job n. Hence, we get that $C_n(\mathsf{pft}) = s_n(\mathsf{pft}) = \frac{1}{2}p_n = \frac{1}{2}k^n$. Observer that at time $\frac{1}{2}k^n$ all other jobs are already finished, so that $s_i(\mathsf{pft}) = 0$ for all $i \neq n$. Combining these observations we get that

$$v(\mathsf{pft}) = w_n s_n(\mathsf{pft}) = \tfrac{1}{2}k^n + \tfrac{1}{2}k^n\epsilon. \tag{31}$$

Second, consider the following solution, which we claim is optimal, where the jobs are processed in order of their index. As all release dates are equal to

zero, we immediately get the time spend on SUB from Theorem 4.3, i. e., we get $s_i(\mathsf{opt}) = \frac{k^i}{2} - \sum_{j=1}^{i-1} \frac{k^j}{2^{i-j+1}}$ for all jobs $i \in \mathcal{N}$. Using this observation, we get the following total value for the optimal schedule

$$\begin{aligned}
v(\mathsf{opt}) &= \sum_{i\in\mathcal{N}} w_i s_i(\mathsf{opt}) \\
&= \sum_{i=1}^{n} \left(\frac{k^i}{2} - \sum_{j=1}^{i-1} \frac{k^j}{2^{i-j+1}} \right) k^{n-i} + \left(\frac{k^n}{2} - \sum_{j=1}^{n-1} \frac{k^j}{2^{n-j+1}} \right) \epsilon \\
&\geqslant \sum_{i=1}^{n} \frac{k^n}{2} - \sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{k^{n+j-i}}{2^{i-j+1}} \\
&= \frac{n}{2} k^n - \sum_{i=1}^{n-1} (n-i) \frac{k^{n-i}}{2^{i+1}}. \qquad (32)
\end{aligned}$$

For the ratio between pft and opt we get from equations (31) and (32) that

$$\frac{v(\mathsf{pft})}{v(\mathsf{opt})} \leqslant \frac{\frac{1}{2}k^n + \frac{1}{2}k^n\epsilon}{\frac{n}{2}k^n - \sum_{i=1}^{n-1} (n-i)\frac{k^{n-i}}{2^{i+1}}} \to \frac{1}{n} \text{ as } \epsilon \to 0, k \to \infty,$$

which shows that the approximation ratio is indeed tight. □

4.5 POLYNOMIAL TIME APPROXIMATION SCHEME

In this section we present a Polynomial Time Approximation Scheme (PTAS) for the case when all release dates are zero. We say that a job is *preempted* if it is scheduled on SUB, but is interrupted before it is completed. We first show that there is always an optimal schedule opt that exhibits some nice properties.

Lemma 4.24. *There always exists an optimal schedule* opt *such that there is no idle time on* SUB.

Proof. Assume there would be an optimal schedule with idle time on SUB. Clearly, all intervals on SUB can be pushed "back" in time, until there is no more idle time between intervals or before the first interval. This action does not change the value of the solution. We now claim that there can also be no idle time on SUB after the last interval. Otherwise, it would be possible to schedule additional time on SUB, thus increasing the total value. This would contradict the fact that

we have an optimal solution, showing that there is always an optimal solution without idle time on SUB. □

Lemma 4.25. *There always exists an optimal schedule opt such that there is at most one interval on* SUB *for each job.*

Proof. To see why this is true, assume there would be a job with at least two intervals on SUB. Fix the last interval for this job. By pushing all other intervals for this job "forward" in time, and any intermediate intervals of other jobs "back" in time, a single interval is created without changing the total value. Note that pushing intervals back is always feasible, since all release dates are zero. □

This lemma implies that for each job i we can define an interval $[k_i, \ell_i)$ during which it is scheduled on SUB. This leads to the following observation.

Observation 4.26. *If job* i *is not preempted, we get* $\ell_i = k_i + \frac{1}{2}(p_i - k_i) = \frac{1}{2}(p_i + k_i)$ *and* $s_i(opt) = \ell_i - k_i = \frac{1}{2}(p_i - k_i)$. *If we increase (decrease)* k_i *with* ϵ, *then* ℓ_i *increases (decreases) with* $\frac{1}{2}\epsilon$ *and* $s_i(opt)$ *decreases (increases) with* $\frac{1}{2}\epsilon$.

Lemma 4.27. *There always exists an optimal schedule opt such that no job is preempted.*

Proof. To the contrary, assume that there is a job that is preempted. Let opt denote an optimal schedule respecting Lemmas 4.24 and 4.25, for which the time point of last preemption is maximized. Let i denote the last preempted job. Clearly, job i is not the last job on SUB, as otherwise its value, and thus also the total value, could easily be increased by not preempting i. Let $\mathcal{N}_i$ denote the jobs scheduled on SUB after i, and without loss of generality assume these jobs are indexed by $1, \ldots, m$. Consider solution fs_1, obtained by increasing the time spend on SUB for job i with a sufficiently small ϵ, and not preempting the subsequent jobs. Similarly, we construct fs_2 by decreasing the time on SUB for i with ϵ. By applying Observation 4.26 we obtain

$$s_i(fs_1) = s_i(opt) + \epsilon, \qquad s_j(fs_1) = s_j(opt) - \left(\tfrac{1}{2}\right)^j \epsilon, \text{ for } j \in \mathcal{N}_i,$$
$$s_i(fs_2) = s_i(opt) - \epsilon, \qquad s_j(fs_2) = s_j(opt) + \left(\tfrac{1}{2}\right)^j \epsilon, \text{ for } j \in \mathcal{N}_i.$$

With respect to the total value obtained by fs_1 and fs_2 we get

$$v(\mathsf{fs}_1) = v(\mathsf{opt}) + \left(w_i - \sum_{j=1}^{m} \left(\tfrac{1}{2}\right)^j w_j\right) \epsilon,$$
$$v(\mathsf{fs}_2) = v(\mathsf{opt}) - \left(w_i - \sum_{j=1}^{m} \left(\tfrac{1}{2}\right)^j w_j\right) \epsilon.$$

If we add both equations, we get that $v(\mathsf{fs}_1) + v(\mathsf{fs}_2) = 2v(\mathsf{opt})$. Since opt is an optimal schedule, it must hold that $v(\mathsf{opt}) = v(\mathsf{fs}_1) = v(\mathsf{fs}_2)$. But then, fs_1 is an optimal schedule for which the time point of last preemption is later than for opt. This contradicts our assumption on opt, which means that there is an optimal solution in which no job is preempted. □

As a consequence of Lemmas 4.24, 4.25, and 4.27, there is an optimal solution opt that can be represented by an ordering of the jobs. Furthermore, for each job i that is scheduled on SUB we can define an interval $[k_i, \ell_i)$ during which it is scheduled on SUB. By definition (see Observation 4.26) we have that $\ell_i = \frac{1}{2}(p_i + k_i)$.

Observation 4.28. *If job i is not scheduled on* SUB*, then for all jobs j that are scheduled before time p_i we have that $w_j \geqslant w_i$.*

Proof. Suppose it does not hold, i. e., there is a job j scheduled before time p_i with $w_j < w_i$. If we now schedule job i instead of j on the interval $[k_j, k_j + \epsilon)$ for some $\epsilon > 0$, we increase the total value. This contradicts that we had an optimal schedule. Hence, there cannot be a job j scheduled before time p_i with $w_j < w_i$. □

Lemma 4.29. *For all pairs of scheduled jobs i and j with $w_i > 2w_j$, i is scheduled before j.*

Proof. To the contrary, assume that the lemma does not hold. Hence, there is a pair of jobs i and j with $w_i > 2w_j$ such that j is scheduled before i. If we now schedule an $\epsilon > 0$ amount of job i on the interval $[k_j, k_j + \epsilon)$, we increase the time spend on job i with $\epsilon/2$, while the time spend on job j decreases with ϵ. But since $w_i > 2w_j$, this has a strictly positive effect on the total value. This contradicts that opt is an optimal schedule, so that the lemma holds. □

Based on these insights the idea for the PTAS is the following. We create disjoint blocks of jobs such that the weights within each block are close to each other, but differ by at least a factor of two from the weights in the other blocks. Lemma 4.29 ensures that in opt all scheduled jobs of a "heavier" block are processed before any scheduled job of a "lighter" block. Observation 4.28 states that we rather process an available job of a heavier block than starting earlier with a lighter block. These two observations allows us to consider each block independently from the heaviest to the lightest, determining the optimal solution for each block based on the remaining processing times. To furthermore decrease the complexity, we round the processing times, and only consider a fraction of the jobs with equivalent processing times. This results in a constant number of jobs per block, for which we try all possible orderings. For each step in this process the loss in total value is limited to a small factor.

Theorem 4.30. SUBCONTRACTOR *admits a PTAS if release dates are zero.*

Proof. Without loss of generality assume that the lowest weight is equal to one, and let $w_{\max}$ denote the highest weight. Divide the jobs according to their weight into groups of bounded ratio $G_1, \ldots, G_\ell$, where $\ell = \lceil \log_2 w_{\max} \rceil$ and $G_j = \{i \in \mathcal{N} \mid w_i \in [2^{j-1}, 2^j)\}$ for $j = 1, \ldots, \ell$. Take an $\epsilon > 0$, and let $k = \lceil \frac{1}{\epsilon} \rceil$. Take an arbitrarily selected j from $1, \ldots, k$, and delete every k^{th} group, starting from G_j onwards. Since all jobs are in exactly one group, this deletion results in an expected $\frac{1}{k} \leqslant \epsilon$ fraction loss in total value. This procedure leaves us with blocks of at most $k-1$ consecutive groups. Note that all the weights in a block are at least twice as high as all the weights in the previous block. As a result of Observation 4.28 and Lemma 4.29, all blocks can be considered independently, starting with the block with the highest weights.

Consider an arbitrary block b, and let $w_{\min}$ and $w_{\max}$ denote the minimum and maximum weight in this block. Observe that by definition, $2^{k-1} w_{\min} \geqslant w_{\max}$. Let $\mathcal{N}_b$ contain the jobs in this block, and rescale the processing times such that the maximum processing time is one. Let opt^b denote the optimal schedule restricted to jobs in this block, and assume without loss of generality that the jobs are processed according to their index. Observe that the total value for this block is at least $\frac{1}{2} w_{\min}$, since the time spend on SUB is at least $\frac{1}{2}$, and the weight obtained at each time point is at least $w_{\min}$. Round down all processing times to

the nearest multiple of $2^{-k}\epsilon$, i. e., $p_i^* = \lfloor \frac{p_i}{2^{-k}\epsilon} \rfloor 2^{-k}\epsilon$ for all jobs $i \in \mathcal{N}_b$, where p_i^* denotes the rounded processing time. We construct a feasible schedule fs for this rounded instance as follows. For the first job, we take the same solution as in opt^b, except that any part processed in the interval $(C_1(opt^b) - \Delta_1, C_1(opt^b)]$ is dropped, where Δ_1 is the difference between the original and rounded processing time. All subsequent jobs on SUB are brought "back" in time with the same amount that is removed from SUB for the first job. For the second job, again any part that is processed in the interval $(C_2(opt^b) - \Delta_2, C_2(opt^b)]$ is removed, and subsequent jobs are brought back in time. This procedure is repeated for all jobs. More precisely, let Δ_i and μ_i respectively denote the amount with which the processing time and the time spend on SUB of job i is decreased, i. e., $\Delta_i = p_i - p_i^*$ and $\mu_i = s_i(opt^b) - s_i(fs)$. Then, fs is defined iteratively for all jobs $i \in \mathcal{N}_b$ as

$$fs_i(t) := \begin{cases} 0 & \text{if } t \in (C_i(opt^b) - \Delta_i, C_i(opt^b)] \\ opt_i^b(t + \sum_{j=1}^{i-1} \mu_j) & \text{else.} \end{cases}$$

As a consequence, we have that $\mu_i = \max\{0, \Delta_i - \sum_{j=1}^{i-1} \mu_j\}$ for all $i \in \mathcal{N}_b$.

Lemma 4.31. $\sum_{i=1}^{m} \mu_i \leqslant 2^{-k}\epsilon$, *for* $m = 1, \ldots, |\mathcal{N}_b|$.

Proof. Observe that by definition we have that $\Delta_i \leqslant 2^{-k}\epsilon$ for all $i \in \mathcal{N}_b$. We will show the lemma by induction. Taking $m = 1$ we get $\mu_1 = \max\{0, \Delta_1\} = \Delta_1 \leqslant 2^{-k}\epsilon$. Next, assume that the lemma holds for $m = 1, \ldots, j$, and consider $m = j + 1$. We get

$$\sum_{i=1}^{j+1} \mu_i = \sum_{i=1}^{j} \mu_i + \max\left\{0, \Delta_{j+1} - \sum_{i=1}^{j} \mu_i\right\} = \max\left\{\sum_{i=1}^{j} \mu_i, \Delta_{j+1}\right\} \leqslant 2^{-k}\epsilon,$$

where the first equality follows from the definition of μ_i, and the inequality follows from the fact that both terms are at most $2^{-k}\epsilon$. □

If we now consider the total value obtained by fs, we get

$$\begin{aligned} v(\mathsf{fs}) &= \sum_{i \in \mathcal{N}_b} w_i s_i(\mathsf{fs}) \\ &= \sum_{i \in \mathcal{N}_b} w_i s_i(\mathsf{opt}^b) - \sum_{i \in \mathcal{N}_b} w_i \mu_i \\ &\geqslant v(\mathsf{opt}^b) - 2^{-k} \epsilon w_{\max} \\ &\geqslant v(\mathsf{opt}^b) - \tfrac{1}{2} \epsilon w_{\min} \\ &\geqslant (1-\epsilon) v(\mathsf{opt}^b). \end{aligned}$$

The first inequality follows from Lemma 4.31 and the fact that $w_i \leqslant w_{\max}$. The second inequality follows from the fact that $2^{k-1} w_{\min} \geqslant w_{\max}$, while the last inequality is implied by the observation that $v(\mathsf{opt}^b) \geqslant \frac{1}{2} w_{\min}$. This means that after rounding the processing times the total value for this block is decreased by at most a fraction ϵ. Moreover, at most $2^{-k}\epsilon$ distinct processing times remain for this block.

From this block b take an arbitrary rounded processing time p. Let $\mathcal{N}_p$ denote the jobs in this class p, i. e., $\mathcal{N}_p$ contains those jobs from block b that have a rounded processing time of exactly p. Without loss of generality assume that $\mathcal{N}_p = \{1, \ldots, m\}$, for some $m \leqslant n$, and that the jobs in this class p are processed in order of their index. Let opt^p denote the optimal schedule restricted to the jobs in class p. Observe that the scheduled jobs in this class can be interchanged without any consequences for the scheduled times of other jobs. As the time a job is processed on SUB decreases the later it is scheduled, scheduling the jobs in non-increasing order of their weight is optimal. This results in the following observation.

Observation 4.32. *$w_{i+1} \leqslant w_i$ and $s_{i+1}(\mathsf{opt}^p) \leqslant \frac{1}{2} s_i(\mathsf{opt}^p)$, for $i = 1, \ldots, m-1$.*

Let fs denote the feasible schedule that is obtained by considering only the k highest weight jobs for processing time p. In this case we have $s_i(\mathsf{fs}) = s_i(\mathsf{opt}^p)$

for $i = 1, \ldots, k$, and $s_i(fs) = 0$ for $i = k+1, \ldots, m$. Combining these observations we get

$$\begin{aligned} v(fs) &= v(opt^p) - \sum_{i=k+1}^{m} w_i s_i(opt^p) \\ &\geqslant v(opt^p) - \sum_{i=k+1}^{m} \left(\tfrac{1}{2}\right)^{i-k} w_k s_k(opt^p) \\ &\geqslant v(opt^p) - w_k s_k(opt^p) \\ &\geqslant \left(1 - \tfrac{1}{k}\right) v(opt^p) \\ &\geqslant (1-\epsilon)\, v(opt^p), \end{aligned}$$

where the first three inequalities follow from Observation 4.32. As the processing time was chosen arbitrarily, it holds for all (rounded) processing times for this block. This implies that by considering the k jobs with the highest weight per processing time class reduces the total value of the block with at most a factor ϵ. Moreover, at most $2^{-k}\epsilon k$ jobs remain for this block, for which we can try all possible orderings.

As the block was chosen arbitrarily, this holds for all blocks. Combining all observations, we can obtain a solution whose total value is at least $(1-\epsilon)^3$ times the optimal total value. Therefore, for any desired approximation guarantee $\hat{\epsilon}$ we can set $\epsilon = \hat{\epsilon}/3$ to get the desired outcome. Since there are at most n non-empty blocks, the running time of this algorithm is $O\left(n 2^{2^{(1/\hat{\epsilon})} \cdot O(1/\hat{\epsilon})}\right)$. Note that it is possible to derandomize the selection process for which group to discard first, by trying all of the first k groups as starting point. This imposes an additional factor k on the running time. □

4.6 EXPERIMENTAL RESULTS

In this section we evaluate several heuristics, among them the ones described in Section 4.4. We first describe how the instances are generated, followed by the heuristics and settings that are considered. The section concludes with a description of the results.

4.6.1 Setup

The heuristics are implemented in C++, run on a machine with an Intel Core 2 Duo E8400 3.00 GHz processor and 4 GB RAM, and evaluated on randomly generated instances.

The instances depend on three parameters: the number of jobs, the distribution of the processing times, and the distribution of the weights. For the number of jobs we consider 10 possible values, starting with 25 jobs and doubling each time to a maximum of 12800 jobs. For the distribution of the processing times we consider three different settings, called Peq, P50, and P75. For Peq all processing times are taken uniformly from the interval $[10.000; 1000.000)$. For P50 and P75 the processing times are split into two sets: the short processing times, taken uniformly from the interval $[10.000; 100.000)$, and the long processing times, taken uniformly from $[100.000; 1000.000)$. For P50 each job has a 50% chance of having a short processing time, while for P75 the chance is 75%. With respect to the weights we consider the exact same three distributions as for the processing time. In this case they are called Weq, W50, and W75. Hence, for Weq all weights are taken uniformly from the entire interval, while for W50 and W75 the weights are split into the light and heavy weights. As before, there is a 50% chance for W50 that a job has a light weight, while for W75 the chance is 75%.

For an overview of the possible values for these parameters, which are called *Jobs*, *ProctimeType*, and *WeightType*, respectively, see Table 22.

Jobs	ProctimeType	WeightType
J25, J50, J100, J200, J400, J800, J1600, J3200, J6400, J12800	P75, P50, Peq	W75, W50, Weq

Table 22: Overview of possible parameter values, sorted from smallest to largest

Based on these parameter values, instances are created in the following way. We first consider the case where Jobs has value 12800. For each job a random value is determined for the release date, processing time, and weight. The release date is taken uniformly from the range $[0.000; 1000.000)$, while the value for the process-

ing time and weight is determined according to the setting for ProctimeType and WeightType. Note that there is no activity before the earliest release date. Therefore, the release date of the first job is set to zero, to remove any potential idle time before the first job is released.

Second, consider the case where Jobs has a value smaller than 12800. In this case, the instances are based on the ones where Jobs has value 12800, by taking an appropriate subset of jobs. Hence, to get the instance for Jobs equal to 6400, we take the first 6400 generated jobs, and similarly for all other number of jobs. This procedure is repeated until 100 instances are generated for all combinations of parameter values, resulting in 9000 instances in total.

Besides the four heuristics (Proctime, Weight, Smith, Profit) described in Section 4.4, we consider five additional heuristics. The first one, called *Index*, simply considers the jobs in the order that they are generated. The other four, called *SmithSquare*, *SmithCube*, *ProfitSquare*, and *ProfitCube*, are variants of Smith and Profit. The results in Section 4.4 indicate that ordering on weight gives the best performance guarantee. Therefore, we place more emphasis on the weight for the additional heuristics. For SmithSquare (SmithCube) we achieve this by considering the ratio between the weight squared (weight cubed) and the remaining processing time, opposed to the ratio between weight and remaining processing time for Smith. Similarly we consider the product of the weight squared (weight cubed) and the remaining processing time for ProfitSquare (ProfitCube), opposed to the product of the weight and the remaining processing time for Profit. Heuristics Smith, SmithSquare, and SmithCube belong to the SmithType heuristics. Similarly, Profit, ProfitSquare, and ProfitCube belong to the ProfitType heuristics.

For the heuristics described in Section 4.4, we consider the case where jobs have release dates and preemption is allowed, but only at the release date of another job. Furthermore, when determining the next job to be scheduled on SUB we consider the current situation. For example, for Smith the job with the highest ratio between weight and *remaining* processing time is selected. For evaluating the heuristics we investigate several additional settings. With respect to the release dates, we consider the case with release dates (*RDyes*) and without release dates (*RDno*), where all jobs are released at time zero. Also for preemption we

consider two situations: one where preemption is allowed (*Pyes*) and one where it is not (*Pno*). With respect to selecting the next job we again consider two cases. The first case corresponds to the one described above, where the current situation is considered (*Iterative*). For the second case the initial situation is considered (*Initial*), e. g., Smith now selects the job that maximizes the ratio between weight and the *initial* processing time.

Combining all possible values for these parameters, which are called *ReleaseDateType*, *PreemptionType*, and *OrderType*, respectively, would result in 8 different settings. However, note that preemption is only allowed at release dates of other jobs. As a result preemption cannot occur if release dates are absent. Hence, if ReleaseDateType has value RDno, we only need to consider value Pno for PreemptionType. This results in 6 different settings to be investigated. See Table 23 for an overview. Observe that the setting considered in Section 4.4 corresponds to Case 6.

Case	1	2	3	4	5	6
ReleaseDateType	RDno	RDno	RDyes	RDyes	RDyes	RDyes
PreemptionType	Pno	Pno	Pno	Pno	Pyes	Pyes
OrderType	Initial	Iterative	Initial	Iterative	Initial	Iterative

Table 23: Overview of possible setting values

4.6.2 Results

In this section we describe the results for the 9 heuristics for each of the 6 cases as described in Table 23. We will focus on five statistics: the weighted savings that are obtained, the time for which SUB is busy, the number of jobs that are processed on SUB, the number of segments on SUB, and the running time of the heuristic. As the algorithms and results for ReleaseDateType RDno (Cases 1 and 2) and RDyes (Cases 3 through 6) are very different, they are usually not compared to each other. Furthermore, for ReleaseDateType RDno, the results for Index, Proctime, and Weight are by definition identical for both values of OrderType. For ReleaseDateType RDyes the results for Index and Weight are also

identical for both OrderType values. For Index this is again by definition. For Weight differences can occur in special circumstances, that apparently are not present in any of the instances. Note that based on the values for Jobs, ProctimeType, and WeightType, the instances can be divided into 90 groups, each consisting of 100 instances. For a summary of the results, see Tables 24, 25, 26, 27, and 28. The columns depict the results for Index (I), Proctime (T), Weight (W), Smith (S1), SmithSquare (S2), SmithCube (S3), Profit (P1), ProfitSquare (P2), and ProfitCube (P3).

4.6.2.1 *Weighted Savings*

Over the six cases, we observe some clear differences in the performance of the heuristics. But we also see some behavior that is consistent over all cases. Let us first focus on the consistent behavior. In line with the results from Section 4.4, overall Weight always performs best. If we split the instances according to ProctimeType or WeightType, then Weight is still always best. If we split according to Jobs, Weight is not always best, for J25 to J100 sometimes other heuristics perform better. Where Weight performs very well, Index and Proctime score badly. They always finishing as worst and second worst, even if we split according to Jobs, ProctimeType, or WeightType. Not only do they always score worst, they also obtain savings that are substantially lower than the savings of the other heuristics. Over the six cases, Index and Proctime overall obtain between 36.74% and 42.77% of the savings for Weight. The seventh best heuristic (either Smith or Profit) obtains at least 76.10% of the savings of Weight. With respect to Jobs, Index and Proctime perform relatively better for smaller instances than for larger instances. For instance, for J25, on average 56.33% (for Cases 1 and 2), 72.96% (Cases 3 and 4), and 63.72% (Case 5 and 6) of the savings for Weight are obtained. For J12800 only 33.85% (over all 6 Cases) is obtained on average. If we consider SmithType and ProfitType, then generally SmithType performs better for the smaller instances, while ProfitType performs better for the larger instances. Clearly, the savings are increasing in ProctimeType. However, there are no substantial differences in relative performance. The difference in savings between P75 and P50 ranges between 7.94% and 8.87%. From P50 to Peq the increase is between 4.59% and 6.13%. Similarly for WeightType, we have that the savings are increasing. With respect to the relative performance, we observe

very small differences for Weight, SmithType, and ProfitType, but large ones for Index and Proctime. For W75, W50, and Weq, on average 23.23%, 36.18%, and 56.11% of the savings of Weight are obtained. The increase in savings from W75 to W50 is between 12.05% and 16.04%. The range for W50 and Weq is between 11.45% and 16.80%.

Case 1. As already stated above, overall Weight obtains the highest savings with a value of 832,089. It is followed by SmithCube, SmithSquare, ProfitCube, Smith, ProfitSquare, and Profit. For this case, Profit still obtains a savings of 793,177 (95.32% of Weight), so the differences between these heuristics are relatively small. Over the 90 groups, Weight obtains the highest savings in 63 of them. SmithCube and SmithSquare score best in 26 and 1 groups, respectively. If we split the instances according to Jobs, we observe that SmithCube scores best for J25 through J100, for all others Weight scores best. For J25 and J50, Weight is even the third best heuristic, with SmithSquare obtaining a second place.

Case 2. For this case Weight by definition obtains the same savings as for Case 1 (832,089). It is now followed by ProfitCube, ProfitSquare, Profit, SmithCube, SmithSquare, and Smith. Compared to Case 1, both SmithType and ProfitType perform worse. Where the difference is small for ProfitType (3.34%, 2.32%, and 1.83% for Profit, ProfitSquare, and ProfitCube, respectively), it is substantial for SmithType (20.54%, 12.04%, and 8.50% for Smith, SmithSquare, and SmithCube, respectively). Over the 90 groups, Weight performs best in 72, and SmithCube in 18. If we split the instances according to Jobs, we get that SmithCube scores best for J25, while Weight scores best for all other values. Where the relative difference between Weight and ProfitType is decreasing in Jobs (from 9.37% for J25 to 2.34% for J12800), it is increasing for SmithType (Smith obtains 94.91% of the savings for Weight for J25, but only 71.06% for J12800).

Case 3. Weight obtains a savings of 1,401,102. The second best is ProfitCube, followed by ProfitSquare, SmithCube, SmithSquare, Profit, and Smith. Smith still obtains a savings of 1,262,984 (90.14% of Weight). Over the 90 groups, Weight scores best in 64, while SmithCube, SmithSquare, and Smith perform best in 15, 7, and 4 groups, respectively. If we consider Jobs, we observe that SmithCube scores best, SmithSquare second best, and Weight only third best for J25 and J50

(although Weight still obtains 99.31% and 99.32% of the savings for SmithCube, respectively). For all other values for Jobs, Weight scores best. For J25 through J200, SmithType performs better than ProfitType, but from J400 onwards ProfitType outperforms SmithType.

Case 4. The savings obtained by Weight are identical to Case 3 (1,401,102). The order of the heuristics is also almost the same, with only Profit and SmithSquare swapping positions. Note that both SmithType and ProfitType perform worse than for Case 3, with Smith now only obtaining 79.57% of the savings for Weight. Over the 90 groups, Weight has the highest savings in 65, while SmithCube, SmithSquare, and Smith perform best in 17, 7, and 1 groups, respectively. With respect to Jobs, we observe that SmithCube is best for J25 and J50. Weight is respectively only third and second best for these values, but is best from J100 onwards.

Case 5. Overall, the savings obtained by Weight are 1,513,970. The order of the heuristics is identical to Case 4. Over the 90 groups, Weight performs best in 85, with SmithCube the best in the remaining 5 groups, all for Jobs J25. If we split the instances according to Jobs, we observe that Weight is always best. For J25 through to J100 SmithCube is second best, from J200 onwards ProfitCube is second best.

Case 6. As for all other cases, the highest savings are obtained by Weight, with a value of 1,513,970, the same savings as for Case 5. Also the order of the heuristics is identical as for Cases 4 and 5. For SmithType and ProfitType the savings are lower as for Case 5 (ranging between 4.57% and 13.70%). Weight performs best in 77 groups, while SmithCube scores best in the remaining 13 groups. Although compared to Case 5 SmithCube overall performs worse, it actually performs slightly better for J25 and J50, by 0.78% and 0.32%, respectively. For J12800 the drop in performance is 9.00%. For J25 SmithCube has the highest savings, for all other values of Jobs, Weight obtains the highest savings.

Case	1	2	3	4	5	6
I	305,707	305,707	577,852	577,852	580,618	580,618
T	316,449	316,449	595,685	599,232	597,230	600,948
W	832,089	832,089	1,401,102	1,401,102	1,513,970	1,513,970
S1	810,116	643,717	1,262,984	1,114,920	1,309,999	1,152,140
S2	822,601	723,569	1,326,338	1,241,048	1,395,848	1,306,350
S3	827,144	756,877	1,351,964	1,292,196	1,431,272	1,366,879
P1	793,177	766,691	1,323,839	1,262,399	1,430,166	1,312,028
P2	806,466	787,750	1,352,598	1,310,339	1,460,922	1,379,338
P3	812,229	797,391	1,364,209	1,332,013	1,473,391	1,408,998

Table 24: Overview of results for weighted savings

4.6.2.2 *Time that* SUB *is Busy*

With respect to the time that SUB is busy with working on jobs, the order of the heuristics is basically always identical, even when splitting according to Jobs, ProctimeType, or WeightType. The highest time on SUB is obtained by Proctime, followed by Smith, SmithSquare, SmithCube, Index, Weight, ProfitCube, ProfitSquare, and Profit. The only differences that occur is that Index and Weight sometimes swap positions. Note that this order is as expected. By definition we have that the time on SUB is maximized if the jobs are processed in non-decreasing order of processing time, as for Proctime. Furthermore, a low processing time is advantageous for being selected by SmithType (and most advantageous for Smith), and disadvantageous for ProfitType (and most disadvantageous for Profit). For Index and Weight the processing time almost plays no role, also explaining why their order sometimes changes. With respect to Jobs and ProctimeType the time on SUB is decreasing, while for WeightType the results are generally stable. Finally, note that the differences are generally quite small.

Case 1. For Proctime and Profit the time on SUB is on average 966.72 and 950.17 (out of a possible 1000), respectively. This corresponds to a difference of

only 1.74%. The highest differences are obtained for J25 (8.00%) and J50 (5.21%). For all other parameter values the difference is at most 2.95%.

Case 2. For Proctime the time on SUB is identical to Case 1 (966.72), while for Profit the value is 945.83, a difference of 2.21%. The biggest differences are again observed for J25 (9.27%) and J50 (6.42%). The maximum difference is 3.94% for the other parameter values. Compared to Case 1, the average time is now slightly higher for SmithType (by 0.34%), and slightly lower for Profit (by 0.32%).

Case 3. The time that SUB is busy with processing jobs is 1818.69 and 1796.42 (out of a possible 2000) for Proctime and Profit, respectively. This equates to a difference of 1.24%. The biggest difference is 3.03% for J50.

Case 4. For Proctime and Profit the time on SUB is 1829.31 and 1784.83, a difference of 2.49%. The largest difference occurs again at J50, with 5.38%.

Case 5. The time on SUB is 1819.71 for Proctime and 1792.80 for Profit. This corresponds to a gap of only 1.50%. For J25 we have the largest gap, with 4.72%.

Case 6. The time that SUB is busy with processing jobs is 1830.85 and 1774.42 for Proctime and Profit, respectively. The difference between these values is 3.18%. The difference is at most 8.57%, for J25. Note that the differences for Case 6 are slightly bigger. We think that this follows from the fact that only for this case the time on SUB is by definition maximized by Proctime. Also noteworthy is the complete opposite behavior of Proctime and SmithType on the on hand, and ProfitType on the other hand: where Proctime and SmithType have the lowest and highest values for Case 3 and 6 respectively, it is the other way around for ProfitType. The differences between the four cases are small, with the biggest difference occurring for Profit with 1.24%.

Case	1	2	3	4	5	6
I	953.51	953.51	1800.49	1800.49	1799.30	1799.30
T	966.72	966.72	1818.69	1829.31	1819.71	1830.85
W	953.52	953.52	1800.39	1800.39	1799.27	1799.27
S1	956.78	961.01	1805.52	1817.23	1806.34	1818.59
S2	955.62	958.76	1803.71	1812.18	1804.06	1813.15
S3	955.10	957.62	1802.84	1809.56	1802.88	1810.21
P1	950.17	945.83	1796.42	1784.83	1792.80	1774.42
P2	951.48	948.68	1797.95	1790.23	1795.13	1783.13
P3	951.96	949.89	1798.57	1792.70	1796.04	1787.00

Table 25: Overview of results for time that SUB is busy

4.6.2.3 *Number of Jobs Processed on* SUB

Compared to the time that SUB is busy, we now generally get the same ranking of the heuristics. This is not surprising: as already stated in Section 4.6.2.2 giving priority to jobs with a low processing time leads to SUB being more busy, but is also gives more options to process other jobs. The only big exception to the ranking of the heuristics is Case 6, which will be discussed in more detail below. Where for the time that SUB is busy the differences are small, we now generally see big differences. For Cases 1, 2, and 6, we have that by definition all jobs are scheduled on SUB for Proctime. With respect to WeightType, there are virtually no differences for Index, Proctime, and Weight, the biggest gap being 2.13% (for Index). For ProfitType the values are decreasing, with differences between W75 and Weq ranging from 6.60% to 12.97%. Again, the only exception is Case 6, where not only the differences are bigger, but also the values are increasing. For SmithType the values are increasing, and the differences are substantial (up to 255%).

Case 1. Overall, the average number of jobs scheduled on SUB is by definition highest for Proctime with 2557.5. This number is substantially higher than for all other heuristics. Smith, SmithSquare, and SmithCube process on average 105.30,

78.10, and 65.87 jobs, respectively. Index and Weight process almost an identical number of jobs, with 16.13 and 16.17. Profit, ProfitSquare, and ProfitCube process the least number of jobs, with respectively 12.62, 13.06, and 13.27. Except for Proctime, the percentage of jobs that are scheduled on SUB is decreasing for Jobs. For instance, for Index 6.47 jobs are on SUB for J25 (25.88%), but only 26.11 jobs are on SUB for J12800 (0.20%). For Index, Weight, and ProfitType we observe that more jobs are scheduled on SUB if there are more short jobs. Over these five heuristics, on average 12.81, 14.34, and 15.60 jobs are on SUB for P75, P50, and Peq, respectively. For SmithType we see different behavior: for Peq clearly the least number of jobs are on SUB (97.92 for Smith), followed by P75 (107.15) and P50 (110.82). The reason why the highest number of jobs is processed for P50 is unknown. For Smith, SmithSquare, and SmithCube there are big differences for WeightType: for W75 on average 74.52, 56.59, and 48.33 jobs are on sub, for W50 103.22, 76.63, and 64.71, and for Weq 138.15, 101.07, and 84.58.

Case 2. For Smith, SmithSquare, and SmithCube the numbers are substantially higher, with 1498.44, 992.78, and 780.85 jobs on SUB, respectively, roughly 13 times as many as for Case 1. For Profit, ProfitSquare, and ProfitCube there is a decrease to 8.87, 9.59, and 10.03 jobs scheduled on SUB. As for Case 1, the percentage of jobs on SUB is decreasing, although for SmithType it remains fairly constant from J400 onwards (59.28%, 39.92%, and 31.72% for J400, compared to 58.46%, 38.67%, and 30.35% for J12800). If we split the instances according to ProctimeType, we observe similar behavior as for Case 1, except for SmithType. For this case, the results for P75, P50, and Peq are almost identical. If we consider WeightType, we witness larger differences for SmithType: 990.27, 473.70, and 359.77 for W75, 1384.90, 900.49, and 706.02 for W50, and 2120.15, 1604.14, and 1276.76 for Weq.

Case 3. Contrary to Cases 1 and 2, not all jobs are necessarily scheduled on SUB for Proctime. In fact, the number of scheduled jobs is significantly lower (152.89 compared to 2557.5). For Smith, SmithSquare, and SmithCube on average 87.99, 74.74, and 67.20 jobs are scheduled on SUB. Index and Weight assist an almost equal number of jobs, namely 17.39 and 17.42, respectively. The lowest number of jobs are processed by Profit, ProfitSquare, and ProfitCube, with 12.99, 13.49, and 13.75, respectively. With respect to Jobs, the percentage of jobs scheduled

on SUB is decreasing, even for Proctime only 390.05 (3.05%) jobs are assisted by SUB for J12800. For ProctimeType we see small differences for Index, Weight, and ProfitType (slightly decreasing for Index and Weight, slightly increasing for ProfitType). For Proctime and SmithType we see bigger differences, especially when comparing Peq (123.58 and 53.51) with P75 (172.73 and 94.34) and P50 (162.36 and 82.08). With respect to WeightType we only observe substantial differences for SmithType: 60.56 for W75, 75.83 for W50, and 93.55 for Weq.

Case 4. As for Case 3, not all jobs are processed for Proctime, but the number of jobs on SUB is significantly higher now (2430.66 compared to 152.89). For Smith, SmithSquare, and SmithCube on average 1423.89, 943.51, and 741.50 jobs are scheduled on SUB, also substantially higher than for Case 3. For Profit, ProfitSquare, and ProfitCube, less jobs are processed, with 9.31, 10.09, and 10.59 jobs scheduled on SUB. With respect to Jobs, we again observe a decreasing percentage of jobs on SUB for Index, Weight, and ProfitType. For SmithType the percentage is first decreasing, but stays relative stable from J100 onwards (55.87%, 37.09%, and 29.35%, respectively). For Proctime we actually see an initial increase: from 74.20% for J25 it increases to 95.02% for J400, after which it stays fairly constant at around 95.08%. With respect to ProctimeType we observe small differences for Index, Weight, and ProfitType. For Proctime and SmithType we see bigger differences, although this time P50 has the lowest values (2381.89 and 1016.73), compared to P75 (2439.54 and 1039.19) and Peq (2470.55 and 1052.98). If we consider WeightType, we primarily see differences for SmithType: 941.79, 450.52, and 341.73 for W75, 1310.50, 851.51, and 666.83 for W50, and 2019.38, 1528.49, and 1215.95 for Weq. Also noteworthy: Proctime processes 2421.33 jobs for W50, compared to 2434.65 for W75 and 2436.00 for Weq.

Case 5. With respect to the number of jobs that are scheduled on SUB, we see similar behavior as for Case 3. Proctime is the highest, with on average 183.69 jobs processed by SUB, roughly 20% higher than for Case 3. For SmithType and ProfitType roughly 30% more jobs are scheduled on SUB. Between Index and Weight we now observe a bigger difference, with values of 21.47 and 25.10, respectively. This corresponds to increases of 23.46% and 44.09%. With respect to Jobs, ProctimeType, and WeightType we generally see the same behavior as for Case 3. The biggest difference occurs for ProctimeType. For Index and Weight

the number of jobs on SUB is decreasing substantially, from 25.92 and 29.06 for P75 to 18.06 and 21.93 for Peq. As a result, the order of the heuristics changes for Peq, with Index dropping from sixth place to eighth.

Case 6. For Proctime all jobs (an average of 2557.5) are by definition scheduled on SUB. For SmithType the values are also high (1764.61, 1162.00, and 932.77, respectively). Especially noteworthy is the following result for ProfitType. For Pno, the number of jobs on SUB is higher for Initial (Case 3) than for Iterative (Case 4), by on average 25.51%. For Pyes it is the other way around, the number is higher for Iterative (Case 6) than for Initial (Case 5), by on average 50.06%. With respect to Jobs, we see similar behavior for SmithType as for Case 4, although the percentage of jobs scheduled on SUB now remains stable from J800 onwards. For Index, Weight, and ProfitType the percentage is again decreasing, reaching at most 0.45% for J12800. Noteworthy is that for J25, more jobs are scheduled for Index and Weight (12.24 and 13.10) than for ProfitType (10.48, 11.04, and 11.37), but for J12800 it is the other way around (29.72 and 36.49 for Index and Weight, compared to 57.47, 50.20, and 46.82 for ProfitType). Furthermore, for ProfitType we have that for J25 the lowest number of jobs on SUB is obtained by Profit and the highest by ProfitCube, while for J12800 it is again the other way around. With respect to ProctimeType we observe that the number of jobs on SUB is decreasing for SmithType. Hence, compared to Case 4, where P50 had the lowest values, now Peq has the lowest values. Note that the values for Peq are even substantially lower than for P75 and P50. For instance, for Smith the values are 1800.14, 1776.92, and 1716.76 for P75, P50, and Peq, respectively. Noteworthy is the behavior for ProfitType. Where for Cases 3, 4, and 5 there are no big differences, the values are now increasing. For instance, for Profit the values are 24.37, 27.73, and 32.05, an increase of 31.51%. With respect to WeightType we observe for SmithType the same behavior as for Case 4. For ProfitType we again observe an increase, which we did not see before. For instance, for Profit the values are 24.83, 28.08, and 31.23 for W75, W50, and Weq, respectively, an increase of 25.78%.

Case	1	2	3	4	5	6
I	16.13	16.13	17.39	17.39	21.47	21.47
T	2557.50	2557.50	152.89	2430.66	183.69	2557.50
W	16.17	16.17	17.42	17.42	25.10	25.10
S1	105.30	1498.44	87.99	1423.89	113.19	1764.61
S2	78.10	992.78	74.74	943.51	97.48	1162.00
S3	65.87	780.85	67.20	741.50	88.67	932.77
P1	12.62	8.87	12.99	9.31	16.98	28.05
P2	13.06	9.59	13.49	10.09	17.68	25.92
P3	13.27	10.03	13.75	10.59	18.05	24.98

Table 26: Overview of results for number of jobs processed on SUB

4.6.2.4 *Number of Segments on* SUB

As for Cases 1 to 4 preemption is not allowed, the number of segments is by definition equal to the number of jobs processed by SUB. Therefore, in this section we will only consider Cases 5 and 6.

Case 5. With respect to the difference between the number of jobs and the number of segments on SUB, the heuristics can be divided into three classes. For ProfitType the difference is 4.20%, for Proctime it is 11.20%, and for Index, Weight, and SmithType it is 16.32%. With respect to Jobs the difference is decreasing: from on average 26.05% for J25 to 7.84% for J12800. For ProctimeType we see big differences in behavior. For Proctime (11.15%) and SmithType (16.84%) the increase is fairly constant, while for ProfitType the difference is slightly decreasing (5.93% for P75, 3.88% for P50, and 2.89% for Peq). For Index and Weight the difference is greatly decreasing: 22.81% for P75, 14.93% for P50, and 6.24% for Peq. With respect to WeightType the differences are small.

Case 6. In this case the heuristics can be divided into four classes. For Proctime the number of jobs processed and the number of segments on SUB are almost the same, the difference being only 0.18%. For SmithType the difference is 13.46%,

and for Index and Proctime it is 15.53%. Especially for ProfitType the differences are huge, on average 1387.94%. Hence, where for the number of jobs processed there are no big differences between Index, Weight, and ProfitType, there are big differences with respect to the number of segments between Index and Weight on the one hand and ProfitType on the other hand. With respect to Jobs the difference is generally decreasing, except for ProfitType, where it is increasing. For instance, for Proctime the difference decreases from 27.44% for J25 to 0.03% for J12800, and for ProfitCube it increases from 23.04% to 3976.87%. If we consider ProctimeType, we see similar behavior for Index, Proctime, and Weight as for Case 5. For SmithType and ProfitType the percentage is decreasing (from 14.83% and 1479.92% to 11.54% and 1284.79%). With respect to WeightType we observe constant behavior for Index, Proctime, and Weight. For SmithType the difference is decreasing. For instance, for Smith the differences are 23.33%, 13.14%, and 4.32% for P75, P50, and Peq, respectively. For ProfitType we see the same behavior as for ProctimeType, with the difference decreasing from 1488.98% for W75 to 1292.90% for Weq.

Case	5	6
I	24.75	24.75
T	204.27	2562.04
W	29.06	29.06
S1	131.13	1980.35
S2	114.00	1316.80
S3	104.42	1071.11
P1	17.57	399.48
P2	18.45	390.33
P3	18.91	383.13

Table 27: Overview of results for number of segments on SUB, for the results of Cases 1 to 4 see Table 26

4.6.2.5 *Running time*

Overall, the running time is very low. The maximum observed running time is 906.069 milliseconds (ms), so just under one second. Observe that for Cases 3 to 6 we consider release dates. Hence, for these instances we first need to determine which jobs are released at which release date. This action, which we call *overhead*, takes roughly 50 ms.

Case 1. The overall running time is very low. The average for Index is 0.011 ms, for the other heuristics the running time is between 0.243 and 0.276 ms. This difference follows from the fact that for Index the jobs do not need to be sorted. When splitting the instances according to Jobs, we see a similar behavior. Index has a very low running time, with only 0.041 ms for J12800. For all other heuristics the running time is approximately doubling for each step of Jobs. Even for J12800 the differences are small, with average running times between 1.278 and 1.447 ms. The maximum observed running time is 1.452 ms. With respect to ProctimeType and WeightType, there are no differences.

Case 2. The heuristics can be divided into two classes. On the one hand we have Index, Weight, and ProfitType (with running times between 0.077 and 0.080 ms), and on the other hand we have Proctime and SmithType (between 29.948 and 38.143 ms). It is interesting to note that SmithSquare (38.143 ms) has a higher running time than Smith and SmithCube (33.659 and 29.948 ms, respectively). This split can be explained by how often the next job on SUB needs to be determined. In this case this coincides with the number of jobs processed by SUB (see Section 4.6.2.3). This is further compounded by the average number of jobs that are available at each such time. For instance, for Profit on average only 3037 comparisons are needed to determine the complete schedule, while for Proctime 3,269,124 comparisons are needed, roughly 1000 times as many. If we consider Jobs, we observe the same split. For Index, Weight, and ProfitType the average running time is 0.004 ms for J25, increasing to 0.375 ms for J12800. The maximum running time for this set is 0.464 ms. For Proctime and SmithType the running time for J25 is also low, with 0.005 ms. It increases to 251.175 ms for J12800, with a maximum of 464.404 ms. Especially the growth of the running time is noteworthy: from J1600 to J12800 the running times are 3.900, 15.402, 61.742, and

251.175 ms, respectively. If we consider the number of comparisons for J1200, we see that on average 20,920 are needed for Profit, while 81,913,600 are needed for Proctime, roughly 3900 times as many. With respect to ProctimeType there are no real differences. Also with respect to WeightType the running times are very constant, except for SmithType. For W75, W50, and Weq the running times are 17.934, 30.965, and 52.852 ms, respectively.

Case 3. For the running times there are no big differences. The minimum is obtained for Index with 51.942 ms, and the maximum for Proctime with 52.670 ms, a difference of only 0.728 ms. Note that the majority of the running time is spend on the overhead. Clearly the running time is increasing with Jobs, but even for J1200 the differences are small: for this parameter value the minimum and maximum running times are 54.298 and 60.319 ms, respectively, a difference of only 6.021 ms. With respect to ProctimeType and WeightType there are almost no differences, the biggest differences are observed for Proctime and ProctimeType, with 0.587 ms.

Case 4. For this case we can again split the heuristics into the same two sets as for Case 2. For Index, Weight, and ProfitType we observe very similar behavior as in Case 3, and will thus not be discussed further for this case. For Proctime and SmithType the average running time is 70.053 ms. We see a substantial increase with respect to Jobs, especially for the large instances. For instance, for Proctime the running time is 50.668 ms for J25, slowly increasing to 51.551 ms for J800. After this it increases to 54.847, 60.454, 88.427, and 212.07 ms. Note that the last step is an increase of roughly 140%. If we ignore the 50 ms overhead, the increase is even around 322%. With respect to ProctimeType we also observe an increase, 62.237 ms for P75, 68.516 ms for P50, and 79.406 ms for Peq. With respect to WeightType we see a constant running time for Proctime, but an increasing running time for SmithType: 61.378 ms for W75, 67.694 ms for W50, and 78.817 ms for Weq.

Case 5. There are no big differences between the heuristics. For the average running time the minimum and maximum are 66.844 and 67.412 ms. Therefore, we will only consider the average running time over all heuristics. Clearly, the running time is increasing in Jobs. Initially the increase is small, from 50.670 ms

for J25 to 51.541 ms for J800. Subsequent running times for J1600 to J12800 are 54.624, 58.679, 80.298, and 170.253 ms. Note that the last step is an increase of roughly 112%, and if we ignore the 50 ms overhead, the increase is even around 297%. With respect to ProctimeType, we see that the running time is increasing, 61.066 ms for P75, 66.082 ms for P50, and 74.242 ms for Peq. With respect to WeightType there are virtually no differences. If we compare the running time to Case 3, we observe an average increase of 28.49%, and an increase of 201.92% for J12800. If we ignore the overhead, the differences are even 663.03% and 1781.89%, respectively.

Case 6. For this case we observe the highest values so far. Compared to Case 4, there are some interesting differences. Index and Weight have the lowest running time (65.968 and 67.805 ms). The next lowest is more surprising, namely Proctime with 84.591 ms. Where for Case 4 it performed substantially worse than ProfitType, it now performs a lot better. The main reason seems to be how often the next scheduled job is determined. For Case 4 these values for Proctime and ProfitType were 2430.66 and 10.00, a ratio of roughly 243. For Case 6 the values are 5103.06 and 2552.46, a ratio of roughly 2. Hence, now relatively a lot less checks are needed for Proctime. These big differences in number of checks can be explained by the fact that now preemption is allowed. Hence, determining the next job to be scheduled on SUB now also needs to be performed on release dates of other jobs. The higher running time for ProfitType can now be explained by the fact that a check is more expensive for ProfitType (one to three multiplications) than for Proctime (a single value). The highest running times are obtained for SmithType. With respect to Jobs we see similar behavior as for Case 4. Initially, the increase in running time is small (on average from 50.671 ms for J25 to 52.292 ms for J800), but it increases rapidly for larger values. For instance, for SmithSquare the running times from J800 to J12800 are 53.048, 60.709, 82.863, 175.070, and 545.769 ms. Observe that the last step is an increase of 211.74%. With respect to ProctimeType we see an increasing running time for all heuristics. With respect to WeightType we observe no substantial differences for Index, Proctime, Weight, and ProfitType. For SmithType we again observe an increase in running time.

Case	1	2	3	4	5	6
I	0.011	0.078	51.942	51.847	66.844	65.968
T	0.252	31.685	52.670	72.321	67.412	84.591
W	0.243	0.077	52.080	51.836	67.038	67.805
S1	0.269	33.659	52.449	73.600	67.270	105.611
S2	0.274	38.143	52.412	68.940	67.250	117.386
S3	0.274	29.948	52.381	65.349	67.238	113.902
P1	0.276	0.079	52.095	51.839	67.040	96.540
P2	0.272	0.079	52.088	51.839	67.038	96.342
P3	0.272	0.080	52.089	51.841	67.041	96.847

Table 28: Overview of results for running time (ms)

4.6.2.6 *Comparison of the Results*

Comparing all results with each other, it appears that only the heuristic Weight needs to be considered. Overall it performs best for all six cases, and even if we split the instances according to Jobs, ProctimeType, or WeightType, it usually obtains the highest savings. Also if we look at the running time, we observe that Weight, together with Index, performs best. However, as the running times are quite low, we suggest to also consider SmithCube and ProfitCube. Especially for the smaller instances SmithCube often outperforms Weight. The choice for ProfitCube might be less obvious. Although it never outperforms Weight, generally the differences are quite small, especially for the larger instances. Also, the number of jobs that are scheduled on SUB are generally smaller than for Weight. This implies less switching for SUB, a cost that we did not consider in our problem setting.

Consider the effect of the different setting. Clearly, if jobs have release dates or not, is not something we can control. But assume allowing preemption is something we can decide on, just as selecting an initial or iterative ordering for the jobs. With respect to PreemptionType, we observe that for all heuristics the savings are higher for Pyes than for Pno, the average difference being 4.77%.

Note that this comes at the expensive of an increased running time of on average 43.24%. If we ignore the 50 ms overhead, the increase is even 873.36%. Next, consider OrderType. As already stated, the results for Index and Weight are identical, so we ignore these heuristics for this setting. Proctime performs better for Iterative, but by only 0.61%. Both SmithType and ProfitType perform worse for Iterative, by on average 6.80%. Also with respect to running time Iterative performs worse.

Concluding, we suggest to consider three heuristics, namely Weight, SmithType, and ProfitType. With respect to OrderType we suggest to use Initial, it almost always performs better (and is never worse for the three proposed heuristic) and has a lower running time. With respect to PreemptionType (if it can be chosen) our recommendation is not so clear cut. On the one hand it gives better results (but only by 4.77%), at the expense of an increased running time of 43.24%. Hence, our advice would depend on the amount of time available. If time is critical, we would advice PreemptionType Pno, otherwise our recommendation would be to use PreemptionType Pyes.

4.7 CONCLUSION

We considered the subcontractor scheduling problem. Given a set of jobs, each with a release date, processing time, and weight, the goal is to find a schedule for the subcontractor that maximizes the weighted sum of savings. For the case without release dates and weights it is a known result that scheduling the jobs in non-decreasing order of their processing times gives the optimal solution. We extended this result by showing that this order is still optimal in the presence of release dates. If furthermore weights are added, scheduling in non-decreasing order of processing time is no longer optimal, and three additional heuristics are evaluated. As it turns out, sorting in non-increasing order of weight gives the best results. For the case with weights but without release dates we construct a polynomial time approximation scheme (PTAS). Finally, for multiple settings we evaluate several heuristics on 9000 randomly generated instances. Again scheduling in non-increasing order of weight gives the best results.

With respect to future research we see several options. First of all, we only allow a single subcontractor, that helps each job with the same speed, which is

equal to the speed of the dedicates machines. It would be interesting to investigate the effect of adding additional subcontractors, possibly with speeds that depend on the job, or even depend on the combination of job and subcontractor. Another possible avenue of research is in the direction of solution methods. So far, only heuristics are considered. One possibility would be to improve these solutions using some sort of local search. Another option is to develop an LP based solution methods, possibly combined with column generation.

BIBLIOGRAPHY

[1] E.H.L. Aarts and J.K. Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.

[2] N. Alon, D. Moshkovitz, and S. Safra. Algorithmic construction of sets for k-restrictions. *ACM Transactions on Algorithms*, 2(2):153–177, 2006.

[3] E.J. Anderson. A new continuous model for job-shop scheduling. *International Journal of Systems Science*, 12(12):1469–1475, 1981.

[4] A. Atamtürk and D. Rajan. On splittable and unsplittable flow capacitated network design arc-set polyhedra. *Mathematical Programming*, 92(2):315–333, 2002.

[5] A. Bachman, T.C.E. Cheng, A. Janiak, and C.T. Ng. Scheduling start time dependent jobs to minimize the total weighted completion time. *Journal of the Operational Research Society*, 53(6):688–693, 2002.

[6] C. Barnhart, C.A. Hane, and P.H. Vance. Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems. *Operations Research*, 48(2):318–326, 2000.

[7] D. Berger, B. Gendron, J.-Y. Potvin, S. Raghavan, and P. Soriano. Tabu search for a network loading problem with multiple facilities. *Journal of Heuristics*, 6(2):253–267, 2000.

[8] V. Bharadwaj, D. Ghose, V. Mani, and T.G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, 1996.

[9] D. Bienstock, S. Chopra, O. Günlük, and C.-Y. Tsai. Minimum cost capacity installation for multicommodity network flows. *Mathematical Programming*, 81(2):177–199, 1998.

[10] P. Bonsma and F. Breuer. Counting hexagonal patches and independent sets in circle graphs. *Algorithmica*, 63(3):645–671, 2012.

[11] A. Bortfeldt and F. Forster. A tree search procedure for the container premarshalling problem. *European Journal of Operational Research*, 217(3):531–540, 2012.

[12] B. Brockmüller, O. Günlük, and L.A. Wolsey. Designing private line networks: polyhedral analysis and computation. *Transactions on Operational Research*, 16(1-2):7–24, 2004.

[13] J. Bruno, E.G. Coffman Jr., and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17(7):382–387, 1974.

[14] M. Caserta, S. Schwarze, and S. Voß. Container rehandling at maritime container terminals. In J. W. Böse, editor, *Handbook of Terminal Planning*, volume 49 of *Operations Research/Computer Science Interfaces Series*, pages 247–269. Springer New York, 2011.

[15] M. Caserta, S. Schwarze, and S. Voß. A mathematical formulation and complexity considerations for the blocks relocation problem. *European Journal of Operational Research*, 219(1):96–104, 2012.

[16] M. Caserta and S. Voß. A corridor method-based algorithm for the premarshalling problem. In M. Giacobini, A. Brabazon, S. Cagnoni, G.A. Di Caro, A. Ekárt, A. Esparcia-Alcázar, M. Farooq, A. Fink, P. Machado, J. McCormack, M. O'Neill, F. Neri, M. Preuss, F. Rothlauf, E. Tarantino, and S. Yang, editors, *EvoWorkshops*, volume 5484 of *Lecture Notes in Computer Science*, pages 788–797. Springer Berlin, 2009.

[17] T.C.E. Cheng, Q. Ding, and B.M.T. Lin. A concise survey of scheduling with time-dependent processing times. *European Journal of Operational Research*, 152(1):1–13, 2004.

[18] V. Chvatal. *Linear programming*. Macmillan, 1983.

[19] A.M. Costa. A survey on benders decomposition applied to fixed-charge network design problems. *Computers & Operations Research*, 32(6):1429–1450, 2005.

[20] T.G. Crainic. Service network design in freight transportation. *European Journal of Operational Research*, 122(2):272–288, 2000.

[21] T.G. Crainic, A. Frangioni, and B. Gendron. Bundle-based relaxation methods for multicommodity capacitated fixed charge network design. *Discrete Applied Mathematics*, 112(1-3):73–99, 2001.

[22] T.G. Crainic and M. Gendreau. Cooperative parallel tabu search for capacitated network design. *Journal of Heuristics*, 8(6):601–627, 2002.

[23] T.G. Crainic, M. Gendreau, and J.M. Farvolden. A simplex-based tabu search method for capacitated network design. *INFORMS Journal on Computing*, 12(3):223–236, 2000.

[24] T.G. Crainic and G. Laporte. Planning models for freight transportation. *European Journal of Operational Research*, 97(3):409–438, 1997.

[25] G. Dantzig. *Linear programming and extensions*. Princeton university press, 2016.

[26] M. Drozdowski. *Scheduling for Parallel Processing*. Springer-Verlag, 2009.

[27] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, 1976.

[28] C. Expósito-Izquierdo, B. Melián-Batista, and M. Moreno-Vega. Premarshalling problem: Heuristic solution method and instances generator. *Expert Systems with Applications*, 39(9):8337–8349, 2012.

[29] FedEx, 2015. `http://about.van.fedex.com/our-story/company-structure/corporate-fact-sheet/` [accessed 7 December 2015].

[30] L. Fleischer and É. Tardos. Efficient continuous-time dynamic network flow algorithms. *Operations Research Letters*, 23(3-5):71–80, 1998.

[31] forbes.com, 2013. `http://www.forbes.com/sites/stevedenning/2013/01/21/what-went-wrong-at-boeing/` [accessed 14 February 2016].

[32] L.R. Ford and D.R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations Research*, 6(3):419–433, 1958.

[33] A. Frangioni and B. Gendron. 0-1 reformulations of the multicommodity capacitated network design problem. *Discrete Applied Mathematics*, 157(6):1229–1241, 2009.

[34] A. Frangioni and B. Gendron. A stabalized structured dantzig-wolfe decomposition method. *Mathematical Programming*, 140(1):45–76, 2013.

[35] A. Frangioni and E. Gorgone. Bundle methods for sum-functions with "easy" components: applications to multicommodity network design. *Mathematical Programming*, 145(1-2):133–161, 2014.

[36] M. R. Garey and D. S. Johnson. Computers and intractability: a guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, 1979.

[37] M.R. Garey and D.S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975.

[38] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[39] B. Gendron, T.G. Crainic, and A. Frangioni. Multicommodity capacitated network design. In B. Sansò and P. Soriano, editors, *Telecommunications Network Planning*, pages 1–19. Springer US, 1999.

[40] B. Gendron and M. Larose. Branch-and-price-and-cut for large-scale multicommodity capacitated fixed-charge network design. *EURO Journal on Computational Optimization*, 2(1-2):55–75, 2014.

[41] B. Gendron, J.-Y. Potvin, and P. Soriano. Diversification strategies in local search for a nonbifurcated network loading problem. *European Journal of Operational Research*, 142(2):231–241, 2002.

[42] I. Ghamlouche, T.G. Crainic, and M. Gendreau. Cycle-based neighbourhoods for fixed-charge capacitated multicommodity network design. *Operations Research*, 51(4):655–667, 2003.

[43] V. Guruswami, S. Khanna, R. Rajaraman, B. Shepherd, and M. Yannakakis. Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. *Journal of Computer and System Sciences*, 67(3):473–496, 2003.

[44] M. Hewitt, G.L. Nemhauser, and M.W.P. Savelsbergh. Combining exact and heuristic approaches for the capacitated fixed-charge network flow problem. *INFORMS Journal on Computing*, 22(2):314–325, 2010.

[45] B. Hoppe and É. Tardos. The quickest transshipment problem. *Mathematics of Operations Research*, 25(1):36–62, 2000.

[46] S.-H. Huang and T.-H. Lin. Heuristic algorithms for container premarshalling problems. *Computers & Industrial Engineering*, 62(1):13–20, 2012.

[47] K. Jansen. The mutual exclusion scheduling problem for permutation and comparability graphs. *Information and Computation*, 180(2):71–81, 2003.

[48] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller, J.W. Thatcher, and J.D. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.

[49] N. Katayama, M. Chen, and M. Kubo. A capacity scaling heuristic for the multicommodity capacitated network design problem. *Journal of Computational and Applied Mathematics*, 232(1):90–101, 2009.

[50] J.L. Kennington and C.D. Nicholson. The uncapacitated time-space fixed-charge network flow problem: an emperical investigation of procedures for arc capacity assignment. *INFORMS Journal on Computing*, 22(2):326–337, 2010.

[51] E. Köhler, R.H. Möhring, and M. Skutella. Traffic networks and flows over time. In J. Lerner, D. Wagner, and K.A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 166–196. Springer Berlin Heidelberg, 2009.

[52] E. Köhler and M. Skutella. Flows over time with load-dependent transit times. *SIAM Journal on Optimization*, 15(4):1185–1202, 2005.

[53] B. Kotnyek. An annotated overview of dynamic network flows. Technical Report RR-4936, INRIA, 2003.

[54] J. Labetoulle, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Preemptive scheduling of uniform machines subject to release dates, 1984.

[55] Y. Lee and S.-L. Chao. A neighborhood search heuristic for pre-marshalling export containers. *European Journal of Operational Research*, 196(2):468–475, 2009.

[56] Y. Lee and N.-Y. Hsu. An optimization model for the container premarshalling problem. *Computers & Operations Research*, 34(11):3295–3313, 2007.

[57] J. Lehnfeld and S. Knust. Loading, unloading and premarshalling of stacks in storage areas: Survey and classification. *European Journal of Operational Research*, to appear, 2014.

[58] T.L. Magnanti, P. Mirchandani, and R. Vachani. The convex hull of two core capacitated network design problems. *Mathematical Programming*, 60(1-3):233–250, 1993.

[59] G.L. Nemhauser and L.A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, 1988.

[60] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994.

[61] C.H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1982.

[62] porttechnology.org, 2015. `https://www.porttechnology.org/news/in_pictures_top_5_transhipment_hubs` [accessed 7 December 2015].

[63] C. Raack, A.M.C.A. Koster, S. Orlowski, and Wessäly. On cut-based inequalites for capacitated network design polyhedra. *Networks*, 57(2):141–156, 2011.

[64] G. Reinelt. Tsplib - a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

[65] I. Rodríguez-Martín and J.J. Salazar-González. A local branching heuristic for the capacitated fixed-charge network design problem. *Computers & Operations Research*, 37(3):575–581, 2010.

[66] M. Skutella. An introduction to network flows over time. In W. Cook, L. Lovász, and J. Vygen, editors, *Research Trends in Combinatorial Optimization*, pages 451–482. Springer Berlin Heidelberg, 2009.

[67] W.E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1-2):59–66, 1956.

[68] R. Stahlbock and S. Voß. Operations research at container terminals: a literature update. *OR Spectrum*, 30(1):1–52, 2008.

[69] D. Steenken, S. Voß, and R. Stahlbock. Container terminal operation and operations research - a classification and literature review. *OR Spectrum*, 26(1):3–49, 2004.

[70] supplychaindigital.com, 2011. `http://www.supplychaindigital.com/top10/2497/Top-10:-Shipping-Companies` [accessed 7 December 2015].

[71] Maastricht University, 2016. `https://www.maastrichtuniversity.nl/file/5809/download?token=7INk9_dG` [accessed 7 November 2016].

[72] UPS, 2015. `https://www.ups.com/content/us/en/about/facts/worldwide.html` [accessed 7 December 2015].

[73] G.L. Vairaktarakis. Noncooperative games for subcontracting operations. *Manufacturing & Service Operations Management*, 15(1):148–158, 2013.

[74] G.L. Vairaktarakis and T. Aydinliyim. Centralization vs. competition in subcontracting operations. Technical memorandum number 819, Case Western Reserve University, 2007.

[75] G. Valiente. A new simple algorithm for the maximum-weight independent set problem on circle graphs. In T. Ibaraki, N. Katoh, and H. Ono, editors, *Algorithms and Computation*, volume 2906 of *Lecture Notes in Computer Science*, pages 129–137. Springer Berlin, 2003.

[76] M. van Brink, G. Csapó, and R. van der Zwaan. The subcontractor scheduling problem.

[77] M. van Brink, A. Grigoriev, and T. Vredeveld. The express delivery problem.

[78] M. van Brink and R. van der Zwaan. A branch and price procedure for the container premarshalling problem. In A.S. Schulz and D. Wagner, editors, *Algorithms - ESA 2014: 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 798–809. Springer Berlin Heidelberg, 2014.

[79] S.P.M. van Hoesel, A.M.C.A. Koster, R.L.M.J. van de Leensel, and M.W.P. Savelsbergh. Bidirected and unidirected capacity installation in telecommunication networks. *Discrete Applied Mathematics*, 133(1-3):103–121, 2003.

[80] V.V. Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[81] I. F. A. Vis and R. de Koster. Transshipment of containers at a container terminal: An overview. *European Journal of Operational Research*, 147(1):1–16, 2003.

NEDERLANDSE SAMENVATTING

In dit proefschrift bekijken we drie operationele plannings problemen, die behoren tot het gebied van combinatorische optimalisatie. Combinatorische optimalisatie problemen worden gekenmerkt door een eindig, of een aftelbaar oneindig, aantal mogelijke oplossingen. Het doel is het vinden van de beste oplossing uit deze mogelijke oplossingen. Aangezien er maar een eindig aantal mogelijke oplossingen zijn, kan de beste oplossing bepaald worden door alle mogelijke oplossingen één voor één na te gaan. Echter, voor grote probleemgevallen zal het bekijken van alle mogelijke oplossingen te lang duren om van praktisch nut te zijn. Een oplossing hiervoor is het ontwerpen van efficiëntere algoritmen die de beste oplossing, of een benadering hiervan, bepalen. Zoals reeds eerder aangegeven bekijken we in dit proefschrift drie operationele plannings problemen. Deze problemen zijn geïnspireerd op problemen uit de praktijk. We bekijken deze problemen zowel vanuit een theoretisch als praktisch perspectief.

Vanuit het theoretisch perspectief bekijken we onder meer de berekenbaarheid van het probleem. Als we weten tot welke complexiteitsklasse een probleem behoort, dan geeft dit een indicatie welk type algoritme gebruikt kan worden voor het vinden van een oplossing. Bijvoorbeeld, als een probleem tot de klasse P behoort, dan bestaat er een efficiënt algoritme dat de beste oplossing bepaalt. Als een probleem echter NP-moeilijk is, dan is het hoogst onwaarschijnlijk dat er een algoritme bestaat dat zowel efficiënt is als de beste oplossing bepaalt. In dit geval is het beter om naar een heuristiek of approximatie algoritme te kijken. Het doel van deze algoritmen is het vinden van een goede, maar niet noodzakelijkerwijs de beste, oplossing in een relatief korte tijd. In sommige gevallen is het mogelijk om een garantie te geven voor de kwaliteit van de gevonden oplossing ten opzichte van de optimale oplossing. Deze zogenaamde prestatiegarantie geeft aan dat, ongeacht het probleemgeval, het algoritme een oplossing zal vinden die niet meer dan een factor gelijk aan de prestatiegarantie van de optimale oplossing afligt. Bijvoorbeeld, als voor een minimalisatieprobleem het algoritme een

prestatiegarantie van 2 heeft, dan zal de oplossing van het algoritme hoogstens 2 keer zo slecht zijn als de optimale oplossing.

Vanuit het praktisch perspectief evalueren we de ontworpen algoritmen op realistische, en dus potentieel grote, probleemgevallen. Deze evaluatie vindt plaats op zowel de kwaliteit van de gevonden oplossing als op de tijd die nodig is voor het bepalen van de oplossing. Hiermee kan worden bepaald welke algoritmen het meest interessant zijn voor de praktijk. Het kan voorkomen dat het ene algoritme beter presteert vanuit theoretisch perspectief (een betere prestatiegarantie), terwijl het andere algoritme beter is vanuit praktisch perspectief (betere oplossingen op realistische probleemgevallen). Ook de benodigde tijd is van belang: al vindt een algoritme hele goede oplossingen, als het te lang duurt om deze oplossingen te vinden, zal het algoritme van weinig praktisch nut zijn. Het is dan beter om voor een algoritme te kiezen dat binnen een acceptabele tijd wel een oplossing vindt, al is die van minder goede kwaliteit.

In Hoofdstuk 2 bekijken we het Express Delivery probleem. Voor dit probleem moet een aantal pakketten bezorgd worden. Van ieder pakket weten we de start en eind locatie, het volume, en de deadline voor bezorging. Het doel van dit probleem is om alle pakketten voor hun deadline te bezorgen, tegen de laagst mogelijke kosten. Het vervoeren van de pakketten wordt uitgevoerd door middel van vrachtwagens. Onder de kosten vallen onder andere het uurloon van de chauffeurs, benzineverbruik, aanschaf- en onderhoudskosten van de vrachtwagen, enzovoorts. Het volume van een pakket is over het algemeen een stuk minder dan de capaciteit van de vrachtwagen. Om kosten te besparen kunnen meerdere pakketten in één vrachtwagen geladen worden. Dit zorgt er echter wel voor dat meer tijd nodig is om alle pakketten in de vrachtwagen te bezorgen, waardoor misschien niet meer alle deadlines gehaald worden. Een belangrijk onderdeel van dit probleem is dus bepalen welke pakketten samen in een vrachtwagen vervoerd worden. Als het aantal mogelijke routes voor de pakketten gereduceerd wordt, wordt ook het aantal mogelijkheden om pakketten te combineren gereduceerd, waardoor de moeilijkheidsgraad van het probleem verlaagd wordt. We gebruiken de volgende methode om het aantal mogelijke routes te verminderen: we kiezen een set van locaties en “promoveren” ze tot zogenaamde hubs. Voor alle pakketten staan we vervolgens alleen maar routes toe die een hub als tussenliggende locatie hebben.

Voor het Express Delivery probleem bewijzen we niet alleen dat het NP-moeilijk is, maar ook dat het moeilijk is om een approximatie algoritme te vinden met een prestatiegarantie die logaritmisch is in het aantal pakketten. We vergelijken een aantal oplossingsmethoden. We hebben een tweetal methoden die gebaseerd zijn op linear programming (LP): de eerste is een MIP model waarvan we de beste oplossing nemen die binnen een kwartier gevonden wordt door een LP oplosser (CPLEX), de tweede een algoritme dat de oplossing van de LP relaxatie afrond naar een geldige oplossing. Daarnaast hebben we een aantal heuristieken, die gecombineerd kunnen worden met lokale zoekmethoden. Deze methoden worden geëvalueerd op 240 probleemgevallen die gebaseerd zijn op probleemgevallen uit TSPLIB. Waar voor een groot aantal probleemgevallen het MIP model de beste oplossing geeft, is er ook een aantal probleemgevallen waar binnen een kwartier voor het MIP model een heel slechte, of zelfs helemaal geen, oplossing gevonden wordt. Ons advies is dan ook om een methode die heuristieken en lokale zoekmethoden combineert te gebruiken. Deze methode geeft voor alle probleemgevallen een oplossing die over het algemeen van goede kwaliteit zijn.

In Hoofdstuk 3 bekijken we het Container Premarshalling probleem. Dit probleem speelt op container terminals. Een van de belangrijkste operaties op container terminals is de overslag van containers. Overslag vindt plaats als een container van één vervoersmiddel (bijvoorbeeld een schip, trein, of vrachtwagen) verplaatst wordt naar een ander vervoersmiddel. Voor het Container Premarshalling probleem bekijken we alleen de overslag van een schip naar een ander schip. Deze overlag vindt meestal niet direct plaats, de container wordt eerst tijdelijk opgeslagen in een speciaal gebied, de zogenaamde containerpark. Een van de belangrijkste indicatoren van de efficiëntie van de operaties op een container terminal is de afmeertijd van een schip. De afmeertijd van een schip bestaat uit de tijd die nodig is om de containers te laden en lossen. Een manier om de laadtijd te verminderen is door het uitvoeren van een operatie genaamd premarshalling. Het doel van premarshalling is het herschikken van de containers in het containerpark, zodat een meer wenselijke lay-out behaald wordt, in een zo min mogelijk aantal verplaatsingen. De prioriteit van een container geeft aan wanneer deze geladen moet worden: hoe hoger de prioriteit, hoe eerder de container geladen moet worden. In een meer wenselijke lay-out staan containers

met een hoge prioriteit bovenaan hun stapel, en containers met een lage prioriteit onderaan.

Met betrekking tot een wenselijke lay-out bekijken we twee verschillende varianten. Bij de eerste variant worden alle lay-outs geaccepteerd waar geen container met lagere prioriteit bovenop een container met hogere prioriteit staat. Bij de tweede variant wordt maar één, voorgedefinieerde, lay-out geaccepteerd. Alleen van de eerste variant was bekend dat deze NP-moeilijk is, en dat ook alleen maar als de maximale hoogte van de stapel gelijk is aan het aantal containers. Wij breiden dit uit door te bewijzen dat beide varianten al NP-moeilijk zijn voor een constante stapel hoogte van ten minste zes. Met betrekking tot de oplossingsmethoden voor het Container Premarshalling probleem ligt de aandacht op het gebruik van heuristieken. Voor zover wij weten zijn wij slechts de tweede die als focus een exact algoritme hebben, en de eerste die het algoritme uitgebreid evalueren. Wij hebben een algoritme ontworpen dat probleemgevallen van de eerste variant kan oplossen. Het algoritme is gebaseerd op Branch-and-Price, en geëvalueerd op 960 probleemgevallen. Hiervan worden 945 probleemgevallen binnen een uur opgelost, 895 binnen een minuut, en 680 binnen een seconde.

In Hoofdstuk 4 bekijken we het Subcontractor Scheduling probleem, oftewel het Onderaannemer Plannings probleem. Voor dit probleem is een set van taken gegeven, met een machine specifiek voor iedere taak. Daarnaast is er nog een speciale machine beschikbaar, die de onderaannemer wordt genoemd. Deze machine kan aan iedere taak worden toegewezen. Iedere taak wordt gespecificeerd door een duur, publicatietijd, en gewicht. Gedurende de tijd dat de onderaannemer aan een taak is toegewezen, wordt de taak met dubbele snelheid verwerkt. Dit zorgt voor een eerdere eindtijd van de taak. Als we de reductie in eindtijd vermenigvuldigen met het gewicht van de taak, krijgen we de gewogen besparing van de taak. Het doel van het Onderaannemer Plannings probleem is het vinden van een geldige toewijzing voor de onderaannemer die de som van alle gewogen besparingen maximaliseert. Een toewijzing voor de onderaannemer is geldig als (1) op ieder moment de onderaannemer aan maximaal één taak is toegewezen, en (2) de onderaannemer alleen maar aan een taak is toegewezen tussen de publicatietijd en eindtijd van deze taak.

Voor het probleem zonder publicatietijden en gewichten was bekend dat de optimale oplossing gevonden kom worden door de onderaannemer steeds aan de taak met de kortste (overgebleven) duur toe te wijzen. We breiden dit resultaat uit, door te bewijzen dat deze aanpak optimaal blijft als publicatietijden aan het probleem worden toegevoegd. Als vervolgens ook gewichten worden toegevoegd, dan is er op dit moment geen efficiënt algoritme bekend dat de beste oplossing bepaalt. Sterker, in dit geval weten we niet eens zeker of het Onderaannemer Plannings probleem NP-moeilijk is. Desondanks hebben we onze aandacht gericht op het ontwerpen van approximatie algoritmen. We vergelijken vier algoritmen, die alleen verschillen in de selectieregel voor de volgende taak die aan de onderaannemer wordt toegewezen. De verschillende selectieregels kiezen de taak met (1) de kortste overgebleven duur, (2) het hoogste gewicht, (3) de hoogste ratio van gewicht tot bewerkingsduur, de zogenaamde Smith ratio, of (4) het hoogste product van de overgebleven duur en het gewicht. Met betrekking tot de prestatiegarantie presteert het toewijzen van de taak met het hoogste gewicht verrassenderwijs het beste. De regel die toewijst aan de hand van de Smith ratio is optimaal voor een aanverwant probleem, en de verwachting was dat deze regel het beste zou presteren, ook omdat in dit geval beide dimensies van de taak (duur en gewicht) gebruikt wordt. Voor het evalueren van de algoritmen zijn nog een aantal varianten toegevoegd: één waarbij de volgende taak geheel willekeurig wordt geselecteerd, en vier varianten op regels (3) en (4), waarbij niet het gewicht wordt genomen, maar het gewicht tot de tweede of derde macht. Dit resulteert in negen verschillende varianten, die geevalueerd worden op 9000 probleemgevallen. Ook in dit geval presteert de regel die de taak met het hoogste gewicht selecteert het beste. Opvallend is dat de regel die de taak met de kortste overgebleven duur selecteert maar marginaal beter presteert dan de volgende taak willekeurig te kiezen.

VALORIZATION

According to the Regulation governing the attainment of doctoral degrees [71], knowledge valorization refers to the "process of creating value from knowledge, by making knowledge suitable and/or available for social (and/or economic) use and by making knowledge suitable for translation into competitive products services, processes and new commercial activities". So what does this mean? In my opinion two main questions need to be answered: (1) what are the benefits to society of my research, and (2) how easy is it to (directly) apply the results.

In the remainder of this section we will first go deeper into the overarching theme combining the problems discussed in this thesis, followed by a short description of the three problems. After that, the benefits to society and the applicability will be considered.

Overarching theme The three problems discussed in this thesis have primarily two things in common: (1) they are based on real-life problems, and (2) they are considered both from a theoretical and practical perspective, with an emphasis on the practical perspective. For the theoretical perspective we consider for instance the complexity of a problem or the worst case performance of an algorithm. For the practical perspective we evaluate the performance of the developed algorithms on realistic instances. With respect to the performance we consider both the quality of the obtained solution and the running time of the algorithm. Note that there is actually no compelling reason why the emphasis is on the practical perspective, except for maybe a personal preference and interest for the practical perspective.

If both the emphasis and the preference is on the practical perspective, why do we still extensively consider the theoretical perspective? One of the reasons is that it can provide valuable information for the practical perspective. For instance, it can give an indication on what kind of algorithm to focus on: if a problem is proven to be NP-hard, it is very unlikely that an exact algorithm exists

that is also efficient. If an efficient algorithm is desired (or maybe even required), it is better to focus on for instance heuristics or approximation algorithms. If an exact algorithm is desired, it will (most likely) have an exponential worst case running time. Also, if we have the worst case performance ratio of an algorithm, we have some kind of indication on the quality of the algorithm; it also provides a mean to compare algorithms with each other.

Note, however, that the theoretical and practical performance of algorithms do not have to coincide: it could be that algorithm A has a better worst case performance guarantee than algorithm B, but performs worse when evaluated on actual instances. A downside of the worst case performance ratio is that it does not tell the whole story, as it is a guarantee that has to hold for all possible instances. It is not uncommon that an algorithm generally performs well, expect for a few (or even a single) artificial instance(s). It would actually be more interesting to have some kind of average case performance ratio. However, it is usually not straightforward to define an average case over all possible instances, and even if it is possible, it is generally hard to analyze. Also evaluating the running time is of importance. It could happen that an algorithm has a good performance with respect to the quality of the solution (this could be both from the theoretical and practical perspective), but if the running time is too high, it will still be of little practical use.

Express Delivery problem The first problem discussed in this thesis is the Express Delivery problem. For this problem we are given a set of packages. Each package is characterized by a start location, end location, volume, and a deadline for delivery. The goal of the problem is to deliver all packages before their deadline, at the minimum cost possible. The transportation of the packages is carried out by trucks. For the cost one can think of for instance the hourly wages of the drivers, gasoline usage, purchase and maintenance cost of the trucks, and so forth. Note that the volume of a single package is generally a lot smaller than the capacity of a truck. Hence, to reduce costs one may want to combine as many packages as possible into a single truck. This however increases the time needed to deliver all the packages in this truck, so that potentially not all deadlines are met. Hence, one main problem is to decide on which packages to consolidate in a truck.

Container Premarshalling problem The second problem discussed in this thesis is the Container Premarshalling problem. This problem occurs at container terminals. One of the main operations at container terminals is the transshipment of containers. Transshipment occurs when a container is moved from one means of transportation (for instance a ship, a train, or a truck) to another means of transportation. In this thesis we solely consider the case where containers are transshipped from one ship to another ship. Note that in this case a container is usually not transshipped immediately, but is first temporarily stored in special area, called the container yard. One of the primary indicators of the efficiency of the operations at a container terminal is the berthing time of a ship, which consists of the time needed to unload and load containers. One way to reduce the loading time is through an operation called premarshalling. The goal of premarshalling is to reshuffle the containers in the container yard into a more desirable lay-out, where containers that need to be loaded first are at the top of their stack, and containers that need to be loaded last are at the bottom of their stack. As the loading plan of a ship usually only becomes available a few hours before arrival, the premarshalling operation should be carried out as fast as possible, which is roughly equivalent to minimizing the number of moves.

Subcontractor Scheduling problem The third problem discussed in this thesis is the Subcontractor Scheduling problem. For the subcontractor Scheduling problem we are given a set of jobs with a dedicated machine for each job, and one additional machine, called the subcontractor, that can be assigned to all jobs. Note that although we use the term machine, one can also think of an actual person. Each job is specified by a duration, release date, and weight. During the time that the subcontractor is assigned to a job, that job is processed twice as fast, resulting in an earlier completion time. If we multiply the amount by which the completion time is reduced by the weight of the job, we obtain the weighted savings of the job. The goal of the Subcontractor Scheduling problem is to find a feasible assignment schedule for the subcontractor that maximizes the weighted savings summed over all jobs. An assignment schedule for the subcontractor is feasible if (1) at each moment in time the subcontractor is assigned to at most one job, and (2) the subcontractor is only assigned to a job between its release date and completion time.

Benefits & Applicability The three problems discussed in this thesis can be categorized as operational planning problems. The main benefits obtained for these problems are of an economical nature. By optimizing the routes for the Express Delivery problem, it might be possible to reduce the number of trucks that are needed and the total distance that needs to be traveled. Optimizing the reshuffling of the container yard for the Container Premarshalling problem has two benefits. Firstly, already by having a better lay-out it might be possible to reduce the loading time, and thus also the berthing time, of a ship. Secondly, by optimizing the reshuffling of the yard, the equipment and available space at the container yard can be used more efficiently. By optimizing the assignment of jobs to the subcontractor for the Subcontractor Scheduling problem , the subcontractor can be put to better use, obtaining a higher return for the company employing the subcontractor.

An additional benefit, especially for the Express Delivery problem, is that the improved solution is more environmental friendly. By reducing the number of trucks, and especially by reducing the number of kilometers that need to be driven, the emission of carbon monoxide is reduced. Note that the economical benefits come as a direct consequence of the objectives that we want to optimize for each problem. For the environmental benefit this is clearly not the case. Hence, the sole objective was to minimize the cost, and obtaining a more environmental friendly solution is just a happy coincidence. However, obtaining more "green" solutions is a topic that is becoming more and more attention and importance.

Now that we have considered some benefits, let us consider the (direct) applicability. The problems considered in this thesis are inspired by real-life problems, and the focus is on the practical side. Unfortunately, this does not mean that the results are directly applicable. While the Container Premarshalling problem is very close to the problem experienced in practice, several abstractions are introduced for the Express Delivery problem and the Subcontractor Scheduling problem. For instance, for the Express Delivery problem each truck is only used on a single connection. This basically means that only the routes for the packages are considered, while the routes for the trucks are ignored. Also, the cost of transshipping a package from one truck to another is not considered. For the Subcontractor Scheduling problem, the processing speed of a job is doubled for

the duration that the subcontractor is assigned to it. Hence, it is assumed that the subcontractor is equally skilled at each job. Furthermore, it is assumed that the subcontractor can instantaneously switch between two jobs, i. e., no time is considered when the subcontractor switches from one job to another. Although the results cannot be used directly, they still have merit and can be used as a stepping stone towards solutions tailored to a more specific problem.

If we finally look at the benefits for society, they are not very surprising. Almost all research towards operational planning problems focuses on minimizing the cost, and thus will have the same economical benefits. But one of the requirements is to specify the benefits of *my* research. Therefore, in the remainder of this section, we consider each of the three problems separately, and discuss some of the more surprising results.

Results for the Express Delivery problem As already mentioned before, one important question for the Express Delivery problem is how much consolidation of packages to use. Related to this question is which level of consolidation possibilities to apply: by restricting the set of routes that packages can use, we can lower the level of consolidation possibilities, decreasing the difficulty of the problem. The method we use to reduce the consolidation possibilities is the following: for each problem instance we pick a set of depots, and "promote" them to hubs. For all packages, we restrict the set of routes that they can use to only include routes that visit hubs as intermediate depots.

At first glance, restricting the possible routes for a package seems counter intuitive: if we only consider a subset of all possible solutions, we surely cannot get a better solution, we might even only be able to obtain worse solutions. This observation would be true, if it would be possible to consider all feasible solution. However, it takes an impractical amount of time to consider all feasible solutions, and in our research we limit the available time to solve an instance to fifteen minutes. Note that even for the most restricted set of routes, fifteen minutes is not always enough to obtain the optimal solution. As it turns out, there are instances for which after fifteen minutes a strictly better solution is found for a more restricted set of routes than for the set of all possible routes. Note that this best found solution for the restricted set of routes would also be feasible for the set of all possible routes, it (or a better solution) has just not been found

within the time limit of fifteen minutes. Hence, in case of a limited amount of time available, it is also advantageous to look at a more restricted set of routes, and thus a more restricted version of a problem.

In a previous section we stated that the theoretical and practical results do not always have to coincide. This is for instance the case if we consider heuristics Independent Shortest Path and Sequential Shortest Path. With respect to the theoretical performance, both heuristics are equally good: if we let n denote the number of packages, we can show that in the worst case both heuristics obtain a solution that is n times as expensive as the optimal solution. If we however consider the practical performance, the cost obtained by Independent Shortest Path is on average roughly twice as high as the cost for Sequential Shortest Path, and in the worst case it is higher by a factor of five. Even with respect to the running time Sequential Shortest Path outperforms Independent Shortest Path.

Results for the Container Premarshalling problem While operations at container terminals have been extensively studied, not much research has been done on the premarshalling problem. With respect to a desirable finish lay-out, we consider two variants. For the first variant, called Priority Stacking, all lay-outs that do not require any unproductive moves during the loading phase are allowed. For the second variant, called Configuration Stacking, only a single pre-specified lay-out is allowed. With respect to the complexity of the problem, it was only known that Priority Stacking is NP-hard if the maximum number of containers per stack is at least the total number of containers. As the maximum number of containers per stack is (at most) eight in practice, this did not provide information on the complexity of the problem as it is encountered in practice. We extend the complexity result by showing that both variants are also NP-hard for a fixed stack height of at least six.

With respect to solving the premarshalling problem, the main focus has been on applying heuristics. As far as we know, we are only the second to focus on an exact algorithm, and the first to extensively evaluate its performance. The other exact algorithm is basically evaluated on only two instances. Whereas the other method took 6802 seconds to obtain the optimal solution for one of the instances, our algorithm was able to solve this specific instance within 1 second.

When we started working on the Premarshalling problem, we decided to focus on an exact algorithm that applies Branch-and-Price. However, at that time it was not clear if this approach could actually work. For using this method, several steps need to be performed. The first step is to reformulate the problem, to obtain a so-called Master Problem (MP). This formulation often has many variables, and therefore a problem with only a subset of the variables, called the Restricted Master Problem (RMP), is initially considered. For the RMP the relaxation is solved. As the RMP does not contain all variables, it might "miss" some variables that are used in an optimal solution for RMP. To solve this issue, another problem, called the Pricing Problem (PP), needs to be considered. How the PP looks like depends on the formulation of the MP. By solving the PP it can be determined if a variable is missing from the RMP. If a variable is missing, it is added to the RMP, and the process repeats itself until no missing variables are found. If the found solution to the RMP is integer, we are done. If it is not, branching needs to be applied. Note that the chosen branching rule has an influence on the PP: the previous branching decisions need to be taken into account. If a wrong branching rule is selected, it could happen that the PP becomes difficult to solve when branching decisions are taken into account. Concluding, there are several challenges when applying a Branch-and-Price approach: the MP and the branching rule need to be defined in such a way that the PP can be efficiently solved, also when previous branching decision are taken into account.

Results for the Subcontractor Scheduling problem In the case of no release dates (or equivalently, all release dates equal to 0) and no weights (or equivalently, all weights equal to 1), it was known that the Subcontractor Scheduling problem is solved to optimality by always assigning the job with the lowest (remaining) processing time to the subcontractor. We extend this result by showing that this approach remains optimal even if release dates are added to the problem.

For the problem where furthermore weights are added, we do not know an efficient algorithm that determines the optimal solution. Even stronger, we do not even know if the problem is NP-hard. We decided to focus our attention on heuristics. The only difference in the heuristics that we consider is in the selection rule of the next job that is assigned to the subcontractor. We consider

four selection rules, namely selecting the job with: (1) the lowest (remaining) processing time (called Proctime), (2) the highest weight (Weight), (3) the highest ratio of weight to (remaining) processing time, also known as the Smith ratio (Smith), and (4) the highest value for (remaining) processing time multiplied by the weight (Profit). Proctime is added because it is optimal in the case of no weights, and Weight is considered because analogously to Proctime it only considers one input dimension. Smith determines the optimal solution for a related problem, minimizing the weighted sum of completion times on a single machine.

Our belief was that Smith, or maybe Profit, would have the best performance guarantee, as they consider both the (reamining) processing time and weight to determine the next job. However, as it turns out Weight has the best performance guarantee, obtaining savings of at least two thirds of the optimal savings. The guarantee is (at most) one half for Smith and $1/n$ for Profit, where n is the number of jobs. For Proctime the guarantee is $1/2^{n-1}$, meaning that for 11 jobs the guaranteed savings is less than 0.1% of the optimal savings.

As noted before, the performance guarantee must hold for all instances. Therefore the practical results might give a different view. Besides the four rules already mentioned, we consider five more selection rules: picking the next job at random, and giving higher importance to the weight for Smith and Profit, by either taking the weight squared or weight cubed. As it turns out, the practical results are in line with the theoretical results. Overall Weight performs best, followed by the variants of Smith and Profit where the weight cubed is considered. What is somewhat surprising is the performance of Proctime. It performs only slightly better than picking the next job at random, but a lot less than the next worst algorithm.

CURRICULUM VITAE

Martijn van Brink was born on February 7, 1980 in Heerlen, The Netherlands. In 1997 he received his havo diploma from Eijkhagen College in Landgraaf. In September of the same year he started a Bachelor in Computer Science at the Hogeschool Limburg, which merged into the Hogeschool Zuyd. After successfully completing the Bachelor, he continued with a Master in Computer Science at the Technische Universiteit Eindhoven.

As a next step he started studying Econometrics and Operations Research at Maastricht University in 2006. As a part of the Bachelor degree he spent one semester abroad at The University of Waikato in Hamilton, New Zealand. In 2010 he obtained his Master degree, with distinction, with Operations Research as specialization.

After graduation, Martijn joined the Department of Quantitative Economics as a Ph.D. candidate in September 2010, under the supervision of Dr. Alexander Grigoriev and Dr. Tjark Vredeveld. The results of his research are presented in this thesis. Martijn presented his work at various international conferences and parts of his thesis are published or under review in leading academic journals. In 2016 Martijn started working at ORTEC in Zoetermeer.